Explicitly Integrated Architecture

An Approach for Integrating Software Architecture Model Information with Program Code

Dissertation

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

durch die
Fakultät für Wirtschaftswissenschaften der
Universität Duisburg-Essen
Campus Essen

vorgelegt von
Marco Konersmann, M.Sc.
geboren in Essen

Essen (2018)

Tag der mündlichen Prüfung: 23. März 2018

Erstgutachter: Prof. Dr. Michael Goedicke, Universität Duisburg-Essen

Zweitgutachter: Prof Dr. Ralf Reussner, Karlsruher Institut für Technologie

Abstract

In component-based software engineering software architectures are specified using components, that are interconnected via interfaces. During implementation, specified architectures are realized using architecture implementation languages, typically on the basis of standardized component frameworks. Both, architecture specification and implementation, concern the same subject while focusing on different aspects. The specification focuses on abstract views for design, communication, and analysis. The implementation focuses on the details of an executable system, with dependencies to the execution platform. Both can be seen as different view points on the architecture.

Both view points together describe the complete architecture, but redundant information exist between them, because they are partly overlapping. This makes the complete architecture hard to understand, because multiple sources of information have to be considered and artefacts have to be mapped to each other. This mapping is often undefined. It is therefore hard to validate whether the implemented architecture matches the specified architecture, and it is often unclear how changes in the specifications should be implemented.

This problem can be solved by reducing the number of sources of architectural information with well-defined mappings to different view points. This approach has the following benefits: (a) the *consistency of the architecture implementation and specification are improved*, by defining a single underlying source of information. (b) Architecture implementations and specifications *survive architecture specification and implementation language evolution*, because well-defined mappings exist between architecture languages, and translations can be automated. Therefore specification and implementation migrations are supported. (c) It makes the architecture *easier to understand*, because only one source of information is necessary to understand the architecture, and the mappings between architecture elements and their implementations are well-defined.

The stated problem is subject to existing approaches. Some of these approaches create mappings between higher level program code fragments, such as code files or directories, and components. The granularity of this mapping is not fine enough to map all architectural aspects. Other approaches have no bidirectional mapping, so that the architecture can only be developed or evolved within one representation, typically the specification. Ad-hoc changes of the architecture within the program code, as they are often observed in "hot" phases of the development, cannot be detected or handled with these approaches. Most related work does not consider the differing focus of architecture implementation and specification languages.

This thesis presents an approach that integrates architecture specification information with program code, so that the program code is the only source of information. The program code includes the architectural information of both, the implementation and the specification language. Architecture specifications expressed in specification languages can be derived from the code, ready for design, communication, and analysis. Bidirectional formal mappings between program code structures and specification elements allow for propagating changes in the specification to the code.

The main contributions of this thesis are:

1. a *Model Integration Concept* for integrating model information with program code, including an development and execution framework for bidirectional model-code transformations,

2. a flexible and extensible *Intermediate Architecture Description Language* for covering the different aspects of architecture languages,

3. a generic framework for *architecture model transformations* as formal mappings between architecture specification language and architecture implementation language elements, and

4. the *Explicitly Integrated Architecture Process*, that uses the Model Integration Concept, the architecture model transformations, and the Intermediate Architecture Description Language for integrating architecture specification models with program code that complies to architecture implementation languages.

The approach presented in this thesis helps software developers by increasing the understandability of the software architecture, and therefore supporting the maintainability and evolvability of the software. Implemented software architectures can not only be visualized, but also changed at a high abstraction level. The Explicitly Integrated Architecture Process can be used for newly developed systems and for existing systems.

The concept and implementation presented in this thesis have been evaluated in four case studies. The first case study is a part of the development of the program JACK 3, an e-learning and e-assessment tool, whose predecessor is in active use in multiple educational institutions. The second case study is a translation of the academic common case study for component modelling approaches CoCoME into a Palladio representation, as preparation for a performance analysis. Palladio is an architecture specification language that can be used to analyse quality aspects of architectures. The third case study translates the CoCoME implementation into a UML model, and the fourth case study migrates the architecture implementation of CoCoME to the Java Enterprise Edition. The evaluation has shown that the approach bridges that gap between architecture implementation and specification languages. The approach provides a single source of architecture information and provides bidirectional transformations between architecture implementation and specification languages via formal mappings between program code structures and model elements. It is a generic approach that can be instantiated for multiple languages, and considers the different focus of architecture implementation and specification languages.

Zusammenfassung

In komponentenbasierter Softwareentwicklung werden Software-Architekturen durch Komponenten spezifiziert, die über Schnittstellen verbunden werden. Bei der Implementierung werden die spezifizierten Architekturen mithilfe von Architektur-Implementierungssprachen realisiert, üblicherweise auf der Basis standardisierter Komponenten-Frameworks. Beide, Architektur-Spezifikation und -Implementierung, betreffen dasselbe Thema, fokussieren jedoch unterschiedliche Aspekte. Die Spezifikation beschreibt abstrakte Ansichten für das Design, die Kommunikation und die Analyse. Die Implementierung beschreibt Details eines ausführbaren Systems, einschließlich der Abhängigkeiten zur Ausführungsplattform. Beide können als Sichten auf die Architektur gesehen werden.

Beide Sichten zusammen beschreiben die komplette Architektur, aber sie stellen Informationen redundant dar, da sie in Teilen überlappen. Dies macht es schwierig, die vollständige Architektur verstehen, da mehrere Informationsquellen betrachtet werden, und Artefakte einander zugeordnet werden müssen. Dabei ist die Zuordnung oft nicht definiert. Daher ist es schwierig zu validierten, ob die implementierte Architektur der spezifizierten Architektur entspricht, und es ist oft unklar, wie Änderungen an der Spezifikation implementiert werden sollen.

Dieses Problem kann gelöst werden, indem die Anzahl der Quellen für Architekturinformationen auf eine einzelne reduziert wird und eine wohldefinierte Zuordnung in verschiedenen Sichten existiert. Dieser Ansatz hat die folgenden Vorteile: (a) Die *Konsistenz der Architektur-Implementierungs- und Spezifikationssprachen wird verbessert*, indem eine einzelne zugrundeliegende Informationsquelle definiert wird. (b) Architektur-Implementierungen und -Spezifikationen *überleben die Evolution von Spezifikations- und Implementierungssprachen*, weil wohldefinierte Zuordnungen zwischen Architektursprachen existieren und Übersetzungen automatisiert werden können. (c) Es wird leichter, die Architektur zu verstehen, da ausschließlich eine einzige Informationsquelle benötigt wird, und die Zuordnung zwischen spezifizierten Elementen und der Implementierung wohldefiniert ist.

Existierende Ansätze gehen das beschriebene Problem unterschiedlich an. Einige dieser Ansätze ordnen abstrakte Code-Fragmente, wie Dateien oder Ordner, Komponenten zu. Die Granularität dieser Zuordnung ist nicht klein genug um alle Architekturaspekte zuordnen zu können. Andere Ansätze haben keine bidirektionalen Zuordnungen, sodass die Architektur nur innerhalb einer Repräsentation entwickelt oder evolviert werden kann, typischerweise in der Spezifikation. Ad-hoc-Änderungen der Architektur innerhalb der Programmcodes, wie sie oft in „heißen“ Phasen der Entwicklung zu beobachten sind, können mit diesen Ansätzen nicht erkannt oder behandelt werden. Die meisten verwandten Arbeiten betrachten den unterschiedlichen Fokus von Architektur-Implementierungs- und Architektur-Spezifikationssprachen nicht.

Die vorliegende Arbeit stellt einen Ansatz vor, der Architektur-Spezifikationen mit Programmcode integriert, sodass der Programmcode die einzige Informationsquelle ist. Der Programmcode enthält die Architekturinformationen der Architektur-Implementierungs- sowie der Architektur-Spezifikationssprache. Architekturspezifikationen in entsprechenden Architektur-Spezifikationssprachen können aus dem Code abgeleitet werden, bereit zum Design, zur Kommunikation oder zur Analyse. Bidirektionale formale Zuordnungen zwischen Programmcode-

Strukturen und Spezifikations-Elementen ermöglichen es, Änderungen in der Spezifikation in den Code zu propagieren.

Die wesentlichen Beiträge der vorliegenden Arbeit sind:

1. ein *Model Integration Concept* zur Integration von Modell-Informationen mit Programmcode, einschließlich eines Entwicklungs- und Ausführungsframeworks für bidirektionale Modell-Code-Transformationen,

2. eine flexible und erweiterbare Architektur-Zwischensprache *Intermediate Architecture Description Language*, zur Abdeckung der unterschiedlichen Aspekte von Architektursprachen,

3. ein generisches Framework für *Architektur-Modelltransformationen* als formale Zuordnungen zwischen Elementen aus Architektur-Spezifikationssprachen und Architektur-Implementierungssprachen, sowie

4. der Prozess *Explicitly Integrated Architecture Process*, der das Model Integration Concept, die Architektur-Modelltransformationen und die Intermediate Architecture Description Language nutzt um Architektur-Spezifikationsmodelle mit Programmcode zu integrieren, welcher mithilfe einer Architektur-Implementierungssprache umgesetzt wurde.

Der in der vorliegenden Arbeit vorgestellte Ansatz hilft Softwareentwicklern, indem die Verständlichkeit der Softwarearchitektur erleichtert wird, und unterstützt damit die Wartbarkeit und die Evolvierbarkeit der Software. Implementierte Software-Architekturen können auf einer hohen Abstraktionsebene nicht nur visualisiert, sondern auch geändert werden. Der Prozess Explicitly Integrated Architecture Process kann sowohl für neu entwickelte Systeme, als auch für existierende Systeme genutzt werden.

Das Konzept und die Implementierung, die in der vorliegenden Arbeit vorgestellt werden, wurden an vier Fallstudien evaluiert. Die erste Fallstudie ist Teil der Entwicklung des Programms *JACK 3*, ein Programm zum E-Learning und E-Assessment, dessen Vorgänger von mehreren Lehrinstitutionen aktiv genutzt wird. Die zweite Fallstudie ist eine Übersetzung der verbreiteten akademischen Fallstudie für Ansätze der Software-Architektur-Modellierung *CoCoME* in eine Palladio-Repräsentation, zur Vorbereitung einer Performance-Analyse. Palladio ist eine Architektur-Spezifikationssprache, die für die Analyse von Qualitätsaspekten genutzt werden kann. Die dritte Fallstudie übersetzt die CoCoME-Implementierung in ein UML-Modell und die vierte Fallstudie migriert die Architekturimplementierung von CoCoME in die Java Enterprise Edition. Die Evaluation hat gezeigt, dass der Ansatz die Lücke zwischen Architektur-Implementierungssprachen und Architektur-Spezifikationssprachen schließen kann. Der Ansatz bietet eine einzige Informationsquelle für Architekturinformationen und bietet durch formale Zuordnungen zwischen Programmcode-Strukturen und Modell-Elementen bidirektionale Transformationen zwischen Architektur-Implementierungs- und Spezifikationssprachen. Es ist ein generischer Ansatz, der für mehrere Sprachen instanziiert werden kann, und beachtet den unterschiedlichen Fokus von Architektur-Implementierungs- und Spezifikationssprachen.

Acknowledgements

The many people who stood by my side during the time I was working on this thesis, who supported me, who were teaching me, who became friends and family, deserve my gratefulness. Michael Goedicke opened my eyes for the joy of scientific work. He created an environment in which I could pursue my research interests and learn to teach. From him I also learned about leading and administrating scientific groups. I'd also like to thank Ralf Reussner for his feedback and for his advices regarding scientific life and work in the last years.

My colleagues in the research group S3 supported me with an enjoyable atmosphere. You make the group a great place. Of my colleagues from the research group S3 I want to give special thanks to Moritz Balz, who motivated me to get a position in science, a profession that I enjoy very much, and a road which eventually lead to this thesis. Also thank you very much for your feedback regarding this thesis. I'd like to thank Michael Striewe for our many discussions and all the shared "Rabulistik" throughout the last years. Thank you for constantly sharpening my word-wielding skills, and for our lovely, shared office.

Most of all, I'd like to thank my family and friends for their love, support, and patience in the time they had to share me with my thesis.

Table of Contents

Part I

Introduction and Related Work

1 Introduction

This chapter first introduces the context of this thesis, describes the addressed problem, and motivates its solution. Requirements for a successful solution are stated, and the objective is formulated in Section 1.6, before a road map is sketched in Section 1.7.

1.1 Introduction to the Specification of Software Architectures

Software architecture is a set of the most relevant design decisions in the development of a software [TMD09, p. 58]. It has a great impact on the quality of a software [TMD09, p. 447], [BBC+10, p. 2]. A variety of software architecture specification[1] languages exist [MT00, BBC+10]. A specified software architecture is realized by the software's implementation artefacts, e.g. the program code, configuration data, and execution platforms [TMD09, p. 337].

Specifications of software architectures can be seen as view points on relevant design decisions. The goals of architecture specification are diverse, generally centering on the design, communication, or analysis of the subject of specification [TMD09, Chapter 2].

Despite a lack of a generally accepted definition of software architecture[2], a set of abstract concerns commonly agreed upon for specifying software architecture seems to exist. These include the general structure, often expressed in terms of components and interfaces, connections between structural elements, abstract descriptions of behaviour [MT00, TMD09, BBC+10, SG96], but also quality aspects like performance [BGMO06, BKR09], security [McD02], or reliability [BKBR12]. Many architecture languages have been developed in the academic and industrial context [MLM+13] for modelling these concerns. However, no language is generally usable in all projects. Thus in different project, typically different languages are used. Even within one project, different language may be used for different purposes, or other languages are used throughout the project lifecycle.

1.2 Introduction to the Implementation of Software Architectures

When software is being developed, the architecture is realized in the software artefacts, including the program code, configuration, and platforms [TMD09, Chapter 9]. The goal of the implementation is an executable system. The implementation of software architecture is driven

[1] Architecture specifications in the context of this thesis means any artefact for unambiguous architecture descriptions. This includes formal notations in textual and graphical forms as well as informal documents with semantics that the artefact's stakeholders agreed upon. This especially – but not only – includes Architecture Description Languages (ADLs) [MT00].

[2] The Software Engineering Institute (SEI) of the Carnegie Mellon University (CMU) has an extensive list of definitions for software architecture in their glossary at `http://www.sei.cmu.edu/architecture/start/glossary/index.cfm` (see "Software architecture"), that contains 267 definitions by 2017-11-23.

by industry standards and platforms that define standard elements such as components and interfaces. Automated build tools are used for packaging parts of the software into deployable units, and these units are deployed on standard execution platforms.

Standards and platforms for implementing software architectures usually do not declare specific rules for imperative behaviour descriptions in program code. Therefore in the context of this thesis, only descriptions of structures are considered. Imperative behaviour descriptions in program code, such as the content of operation bodies, are not within the scope of this thesis.

1.3 Motivation for Considering the Differences between Software Architecture Specifications and Implementations

Figure 1.1 shows an example of a specification and an implementation view on an excerpt of a software architecture. Both represent the same information: A component with the name `CashDesk` exists that requires two interfaces `IBarcodeScanner` and `IPrinter`. The implementation adds information unnecessary for the specification, i.e. the package declaration, detailed structure and behaviour information (abbreviated with `[...]`), and dependencies to the underlying execution framework: the component is declared by a Java annotation `Stateful`, meaning this class is a component that does not store a session specific state, and the required interfaces are declared using the annotation `@EJB`, which instructs the execution platform to resolve the dependencies in a specific way.

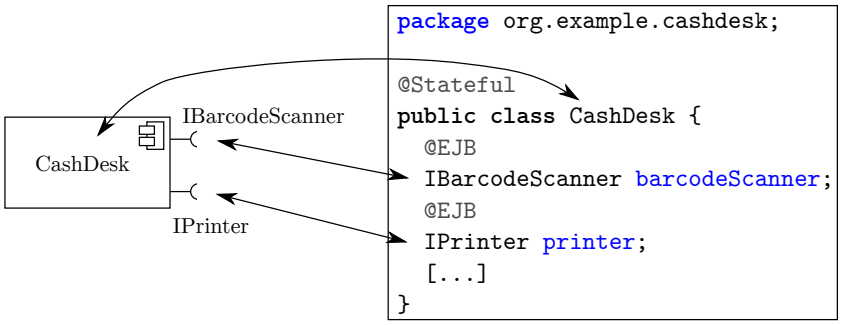


Figure 1.1: An excerpt of a software specification (left) and a corresponding implementation (right)

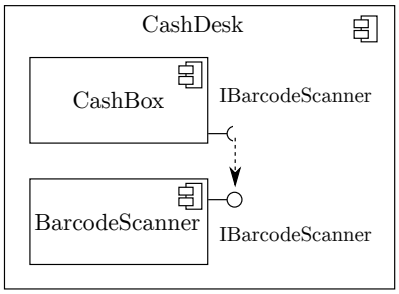
Both, architecture specification and implementation, concern the same subject while focusing on different aspects. The specification focuses on views for design, communication, and analysis. The implementation focuses on the details of an executable system, with dependencies to the execution platform. They are considered different artefacts, and are usually notated in different files or documents. Nevertheless they are strongly coupled, because they share common architectural information. Both can be seen as different views on the architecture. The following section inspects the commonalities and differences between architecture specifications and implementations.

1.4 The Difference between Architecture Specifications and Implementations

Many different architecture specification and implementation languages exist, of which many are devoted to a special architectural style or domain. It is, however, possible to identify commonalities and differences [Mü10, MBG10, MT00]. As an assumption, this thesis distinguishes between two types of commonalities:

Explicit commonalities have first class elements in both the specification and the implementation. E.g. in Figure 1.1 a specified component is implemented as a Java class using an annotation. Explicit commonalities are typically the representation of components, interfaces, and connectors [MBG10].

Specifications that are translated into composed implementation structures have first class elements in a specification, that can be mapped to composed structures in the architecture implementation, which are considered equivalent. As an example, Figure 1.2 shows an architecture with a hierarchical component specification. The component `CashDesk` contains the components `CashBox` and `BarcodeScanner`. The implementation of this architecture uses Java packages as hierarchical namespaces to model the parent-child relationship. While the namespaces represent the relationship, they do not carry semantics included in the relationship. I.e. the component `CashBox` could reference any component outside its parent in the implementation, while this is not allowed in the specification.




```java
package org.example.cashdesk;

@Stateful
public class CashDesk { [...] }
```

```java
package org.example.cashdesk.cashbox;

@Stateless
public class CashBox { [...] }
```

```java
package org.example.cashdesk.barcodescanner;

@Stateless
public class BarcodeScanner { [...] }
```

Figure 1.2: A specification (left) and an implementation (right) of a component hierarchy. The implementation uses the package hierarchy to represent the parent-child relationship.

This thesis distinguishes between two types of differences:

Specification details that have no representation in the implementation are elements in the specification, that have no equivalent element or structure in the implementation. An example for such information is the annotation of subsystems with quality properties. While these annotations can be used for a quality analysis, they typically have no representation in the implementation.

Implementation details that have no representation in the specification are elements in the implementation, that have no equivalent element or structure in the specification. Examples for such information are platform dependencies or detailed behaviour descriptions. While such program code is used to create an executable system, it is too specific for being represented in the abstract specification.

The reason for these differences lie in the goals and presumptions of the view points. The specification does not consider e.g. platform dependencies or constraints due to the reuse of standard libraries. The implementation instead needs to take the platform and adjacent technologies into account, while its main goal is to create an executable system.

1.5 Motivation for Bridging the Gap between Software Architecture Specifications and Implementations

The section above identified a gap between the two view points on the software architecture. Bridging this gap could help with the following concerns:

The consistency of the architecture implementation and specification are improved: When an architecture is defined using multiple views that show different concerns, these views are often in inconsistent states. During maintenance and evolution of a software, the architecture specification and implementation are changed independently from each other.

The specification and implementation in the example in Figure 1.3 have evolved independently. The specification shows a required interface `IPrinter` for the component `CashDesk`, which is not existent in the implementation. In this situation, without further information, it is unclear whether the interface needs to be added in the implementation or removed from the specification. When the views are not kept in an consistent state, the information on the architecture might be misleading.

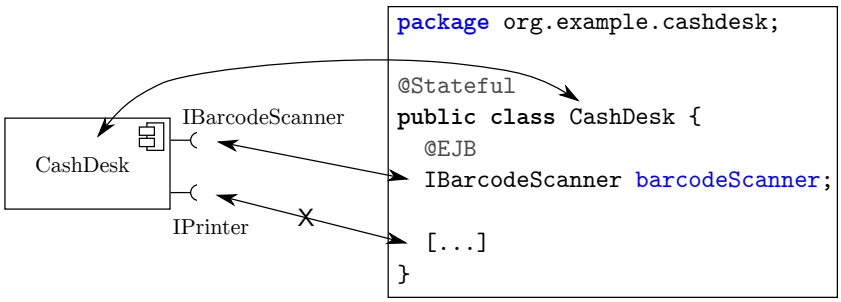


Figure 1.3: Inconsistent software specification (left) and implementation (right)

It is possible to derive the architecture specification from the implementation. Therefor a precise mapping between the two views needs to exist, and the implementation must include all commonalities and all specification details. Using the implementation as the single underlying model for the implementation and the specification integrates the

information of these highly coupled views. It ensures the consistency between the views, which increases the evolvability and maintainability.

The consistency between program code and the architecture specification is also a stated need by practitioners. In a study of Malavolta et al. [MLM+13], 37% of the answers stated that reverse and forward engineering between architecture languages and the implementation would be useful in the future, opposed to 24% who stated it would not be useful. 70% stated that a support for the alignment of software architecture descriptions with the implemented system will be useful in the future.

Architecture implementations and specifications survive language evolution: In long-living software systems it is not uncommon that the languages used to implement or specify architectures changes over time. Many reasons lead to implementation languages and execution platforms to become outdated. Such reasons include the decrease of the number of experts for specific languages when new, better languages emerge or when execution platforms are not maintained any more. Even successful and broadly used languages evolve, and the changes between language revisions can be severe. During an implementation migration the software architecture needs to be reimplemented in the new language. The reimplementation is often expensive, without visible improvements for users, and it can introduce errors.

When a precise, known, and complete mapping between the specification and the implementation of the software architecture exists, it is possible to automatically translate between different architecture implementation and specification languages. This allows to handle **Implementation Language Evolution**: A software architecture implementation can be translated into a specification, and that specification into an implementation in another implementation language. While such a translation would not include implementation details, the new implementation would already include common elements between the implementation and specification languages.

Architecture specification languages also evolve. On the one hand new specification languages may provide new, desired features, on the other hand existing languages may evolve, and corresponding visualization and analysis tools may rely on the use of the most current version of the language. The specification must be migrated to the new language version or the corresponding tools might be unusable.

A precise, known, and complete mapping between the specification and the implementation of the software architecture also allows for handling **Specification Language Evolution**. Such mappings can be used to derive specifications in arbitrary specification languages from the implementation. When a specification language evolves, it is possible to derive a representation of the architecture in the newer language from the implementation, by adapting the original mappings to the new language.

The understandability of the architecture is increased: For understanding the whole architecture, it is necessary to consider all views. In the example in Figure 1.1 the specification and an abstract implementation of a component `CashDesk` is shown. For understanding how the component is implemented, it is necessary to read both the specification and the implementation and to create a mapping. In the example, it is unclear whether a component is implemented using a class in a package named after the specified component, using a class that is named after the specified component, or a combination of both.

When the mapping is not communicated, it will probably not be used consistently. Because the architecture comprises the most relevant design decisions, a misunderstanding or an inconsistent use of this mapping can impose relevant mistakes during development, maintenance, and evolution.

Therefore it is necessary to create a precise mapping between architecture specification elements and architecture implementation elements and make this mapping available to all stakeholders of the software architecture.

1.6 Requirements and Objective of this Thesis

Section 1.4 identified differences between software architecture specification and implementation languages, which lead to a gap between architecture specifications and their implementation. Section 1.5 motivated why this gap should be bridged. The following requirement is therefore formulated for the co-evolution of architecture models and program code:

R1 *Bridge the gap between software architecture specification languages and implementations thereof.*

For evaluating whether R1 is met, the following questions have to be answered:

Q1.1 Does a semantic equivalence relation exist for *explicit commonalities*?

Q1.2 Does a semantic equivalence relation exist for *specifications that are translated into composed implementation structures*?

Architecture implementation and specification languages have commonalities and differences as shown in Section 1.4. R1 considers the commonalities of these languages. The differences are considered with the following requirement:

R2 *Take the differences of architectural specification and implementation languages into account.*

For evaluating whether R2 is met, the following questions has to be answered:

Q2.1 Do program code representations exist for *specification details, that had no representation in the implementation* before?

Q2.2 Are *implementation details, that have no representation in the specification*, preserved during changes in the specification?

When two views upon software architecture overlap, the views must be consistent regarding this overlapping part. I.e. the two views must not define contradictory specifications [LMT09] or specifications that do not fit together, e.g. when a component is used via an interface that it does not provide. If the views are in an inconsistent state, misunderstandings may occur. To avoid inconsistencies between architecture views, for each architecture element there must be exactly one original source of information. Other views may be derived from that origin. The following requirement is formulated:

R3 *Provide a single source of information for architecture descriptions.*

For evaluating whether R3 is met, the following question has to be answered:

Q3.1 Does a single source contain all implemented and specified architecture information?

To also make the architecture changeable within the syntax of specification languages, an approach must provide bidirectional translations between the specification and the implementation of architectures. Therefore the following requirement is formulated:

R4 *Create bidirectional translations between the architecture specification and the implementation.*

For evaluating whether R4 is met, the following questions has to be answered:

Q4.1 Can specification views be derived from program code?

Q4.2 Are changes in the derived specification views propagated to the program code?

In practice many languages for specifying and implementing architectures are used. It can be expected that current architecture specification and implementation languages will continue to evolve in the future, and that new languages will emerge. To make architecture implementations and specifications survive the evolution of languages, a general approach is desirable, which can take current and future languages and their different features into account. This is reflected in the following requirement:

R5 *Prepare for architecture specification and implementation language emergence and evolution.*

For evaluating whether R5 is met, the following questions has to be answered:

Q5.1 Can multiple architecture implementation and specification languages be used with the approach?

Q5.2 Are languages weakly coupled with other languages in the approach?

Related work to this thesis does not bridge the gap sufficiently (see Chapter 3). Therefore a need is identified for bridging this gap, while considering the specific advantages of both view points on the software architecture. The following objective for this thesis is derived:

Develop concepts for bridging the gap
between software architecture specification and implementation

1.7 Thesis Road Map

This thesis is structured into three parts as follows:

Part I – Introduction and Related Work After the motivation and the description of the thesis' goals above, Chapter 2 introduces the conceptual foundations of this thesis. Chapter 3 describes existing related work. It shows – with the requirements stated above – that existing solutions do not sufficiently bridge the identified gap.

Part II – Bridging the Gap between Architecture Specification and Implementation This part describes the research contribution. Chapter 4 gives an overview of the proposed solution, which is used to overcome the challenges described in the motivation. It comprises a *Model Integration Concept*, which is further described in Chapter 5, an *Intermediate Architecture Description Language* described in Chapter 6, and a set of *Architecture Model Transformations*, which are described in Chapter 7. Chapter 8 describes a process how the parts of the proposed solution interact to bridge the gap between architecture specifications and implementations. Chapter 9 gives an overview of the implementation of a set of tools that are a foundation for the development and execution of the process.

Part III – Evaluation and Conclusion Chapter 10 describes the evaluation of the contribution. The prototype described in Chapter 9 is used to bridge the gap between the architecture specification and implementation of existing programs. The conclusion and thoughts on future work are given in Chapter 11.

The appendix of this thesis comprises the list of references, an overview of the contents of the data medium that accompanies this document, examples of the use of the architecture language defined in this thesis, the list of integration mechanisms presented in Chapter 5, lists of figures, listings, definitions, and examples.

2 Conceptual Foundations

In Chapter 1 languages for architectural description and implementation were described as central foundation of this thesis. The chapter at hand lays the basis for understanding this thesis, by presenting further conceptual foundations. Abstraction is a key concept for architectural description and design. Section 2.1 presents the aspects of abstraction in software engineering that are relevant for the topic of this thesis. The concepts described in the following chapters are based on languages with an abstract syntax defined with meta models. Section 2.2 describes meta modelling and its use in model-based software engineering. At last, Section 2.3 gives a brief overview of the languages and tools used during the implementation of the prototype presented in Chapter 9.

2.1 Abstraction in Software Engineering

Abstraction in software engineering means to ignore details that are unnecessary in a given situation [GJM03, p. 49]. Abstraction can be found in many places in the software engineering domain. Some examples are: (a) A programming language is an abstraction of machine code. (b) A software design is an abstraction of a detailed implementation. (c) Requirements are abstractions of a possible solution for a given problem. Even business processes that are supported by automation are abstractions of their underlying problem.

Abstraction in software engineering is used for mastering complexity [GJM03, p. 49]. In a technical sense, abstraction is not necessary for developing software. A program can be functional without any technical abstractions. E.g. a program written in machine code for a specific hardware is technically able to work as intended. Abstraction, such as the use of high-level programming languages, the use of patterns [GHJV95], or architectural design specification are only necessary for humans to increase their efficiency or effectiveness. In the context of software architecture, three relevant kinds of abstractions are views and view types, modularity, and hierarchies.

2.1.1 Views, View Types, and View Points

When a stakeholder takes part in the development of a system, it makes sense not to see the complete system as a whole in all its details. Instead a specific *view* upon the system should be provided, which includes the necessary information but does not contain information unnecessary for the current task [FS96].

View types are the generalization of views. A view type describes the abstract and concrete syntax for describing a view. Kruchten's 4+1 view model [Kru95] is an example for organizing the specification of software architecture using view types. The 4+1 view model comprises a *logical view*, that describes the object model; the *process view*, which describes concurrency aspects; the *physical view* for describing the deployment on hardware; and the *development view*, that describes the static structure. The fifth (+1) view is named *scenarios*, and describes examples how the elements of the other four views are used together. For each of these views

a specific notation exists. Further view types are defined in ISO/IEC/IEEE 42010 [ISO11] (Systems and software engineering — Architecture description) or the Unified Modeling Language (UML) [Obj15]. View-based software engineering is concerned with the development of software using multiple interdependent views. *View points* are sets of view types, which have common abstract syntax elements [The13, Sec. 8.2].

Views in view-based software engineering should be consistent to each other to provide a base for executing a functionally correct software with the desired qualities. E.g. when a system implementation and its requirements are inconsistent, the software probably does not implement the requirements correctly. Views can be kept consistent e.g. by actively managing consistency after any change. This might be supported by an underlying consistency model. However, keeping views in a consistent state is considered a major challenge in view-based software engineering. In the context of this thesis the program code and the architecture specification are considered views upon the software, which have to be kept in a consistent state. Therefore architectural views should be derived from the program code view, and the program code should be changed according to the changes in the architectural views.

2.1.2 Modularity

Modularity [GJM03, p. 47] in software engineering can be used to divide complex systems into smaller pieces. These smaller pieces are called *modules*. Modules are in directed relations with each other, e.g. a module calls another module or a module is part of another module. The details of one module can be developed independently from the other modules, as long as the endpoints of these relationships are not subject to change. Systems can be composed of multiple modules. Such composed systems are called *modular* systems. A modular system can be built in two stages: (a) The internal details of the modules are developed. (b) A system is composed of multiple modules. Depending on the order of the stages another development style is employed: A *bottom-up* development means that first the modules are developed or reused (and probably adapted) from a module repository. Then the system is composed of these existing modules. A *top-down* development means that first a system to be built is decomposed into multiple modules. Then these modules are developed or reused (and probably adapted) from a repository. Parnas describes criteria for decomposing systems into modules in a top-down approach [Par72]. The composition can happen on multiple levels, where composed modules are also composable in a broader context. Two desired properties for modular systems are high cohesion and low coupling. High cohesion means that all elements within a module are related strongly. Low coupling means that the interconnections between modules are weak. Modular systems with a high cohesion and low coupling make it easy to exchange or evolve single modules.

In a more general sense, modularity can be seen as the concept *divide et impera* (divide and conquer), where a problem is divided into multiple smaller pieces which have a limited scope. These "conquered" smaller pieces can then be composed. In the context of this thesis, systems are decomposed into modules. The modules can be implemented with general purpose programming languages. These modules can then be composed using architecture specification languages.

2.1.3 Hierarchies

Modules were described above as elements that are interconnected with directed relations. Hierarchies in this context are directed acyclic graphs of these relations [GJM03, pp. 79-83]. Consider the relation $r \subseteq M$, with M being a set of modules. When a module $M_1 \in M$ calls another module $M_2 \in M$, then $M_1 \xrightarrow{r} M_2$. This relation is transitive. The transitive closure $M_1 \xrightarrow{r^+} M_3$ means that either $M_1 \xrightarrow{r} M_3$ or $M_1 \xrightarrow{r} M_2$ and $M_2 \xrightarrow{r^+} M_3$. A relation is a *hierarchy* when there are no modules M_1 and M_2 so that $M_1 \xrightarrow{r^+} M_2$ and $M_2 \xrightarrow{r^+} M_1$.

In general, hierarchies can be seen as directed acyclic graphs of relations between elements. Within this thesis abstraction hierarchies of architecture views are created. The program code eventually holds all necessary information. Other architectural views upon the program code are derived. These derived views are more focused on architectural aspects and therefore describe the architecture on a more abstract level.

2.2 Modelling, Meta Modelling, and Model-Driven Software Development

A model can be seen as an abstraction of a subject. An example for models can be mathematical formulas that describe the reality, while ignoring factors that are irrelevant for the use case. Models of software are often represented as interconnected elements, e.g. structural models or behavioural models of the UML. Modelling is the activity to create models.

Meta models are the generalisation of model elements. Meta models define the abstract syntax of models that comply with the meta model. A model that complies with a meta model is called an *instance* of the meta model. The key concepts behind meta modelling are the relationship between a model element (often called *object* or *instance* in this context) and its meta model element (*classifier* or *class*) and the ability to navigate from an object to its classifier. Multiple levels of instance-of relationships are possible, where the classifier of an object is itself the instance of a "higher level" classifier. Two *meta levels* mean that one level of objects and one level of classifiers exist. An arbitrary number of meta levels is possible, although typically two to four levels are used [Obj16, Section 7.3]. Instance-of relations in meta modelling build directed acyclic graphs. They form a hierarchy (see Section 2.1.3).

Meta Modelling is the activity to create meta models. This can follow a top-down or a bottom-up approach. Top-down meta modelling means to define a meta model for a subject to model and to create models afterwards. Bottom-up means to derive a meta model out of a modelled subject to classify the already modelled elements.

Model-driven software development (MDSD) [SVC06] uses models as central artefacts for software development activities. In MDSD parts of the software are described using models that comply to domain-specific meta models [MF10]. These domain models are refined with detailed technical models that are not relevant to the domain, but to the platform that will run the software. Such models are the basis for automated code generation. The generated code has to be enriched with implementation details.

2.3 Languages, Standards, and Tools used Within this Thesis

This section briefly describes languages, standards, and tools used within this thesis.

2.3.1 Ecore and the Eclipse Modeling Framework

Ecore [SBPM09, Chapter 5] is a meta meta model, the highest level in a meta model hierarchy of three meta levels. It therefore declares an abstract syntax for describing meta models. The Eclipse Modeling Framework (EMF)[1] [SBPM09] is a set of specifications and tools for building and using domain-specific meta models in Java. It uses Ecore as its foundational meta meta model. Among the features provided by EMF, the following are the most relevant for this thesis:

- Meta models can be created based on Ecore.
- Program code can be generated to create models of the meta models and to interact with them.
- A rich ecosystem of tools exist based on Ecore meta models, such as languages for model transformations and code generation.

In this thesis the meta models of the implemented prototype are technically based on Ecore and built with EMF.

2.3.2 Model Transformations with Henshin

Henshin[2] [ABJ+10a] is a graph transformation [Roz97] language and tool for EMF, with formal graph transformation semantics. It supports two types of transformations: Endogenous transformation of models result in a changed model of the same meta model. Exogenous transformations result in a new model of another target language. It uses a graphical syntax for the definition of transformations. A model transformation definition in Henshin consists of possibly multiple rules and units. Rules describe actual transformations. Units describe the order of rules, repeated, concurrent, or conditional execution. Rules describe the left hand side of a transformation and the right hand side of a transformation in an integrated view.

Figure 2.1 shows a simple example of a rule named `createCashDesk`, expressed with the graphical editor of Henshin. The left hand side of a rule describes which element should be found, and eventually changed, within a model. In the example, the left hand side defines, that an element *Architecture* is searched in the model, which is subject to transformation. Each of these elements found forms a *match* in a list of matches. For each of these matches, a negative application condition is checked: If the architecture element has a reference *componentTypes* towards an element of the type *ComponentType*, which has the value `CashDesk` for the attribute *name*, the match is removed from the list of matches. The right hand side defines, that for each match in the list of matches, a new element *ComponentType* should be created with the value `CashDesk` for the attribute *name*. A new reference *componentTypes* will be created from the architecture element in the match to the newly created component type. Besides the graphical editor and menu items for the Eclipse platform for executing transformations

[1] Eclipse Modeling Framework – `https://www.eclipse.org/modeling/emf/`
[2] Henshin – `https://www.eclipse.org/henshin/`

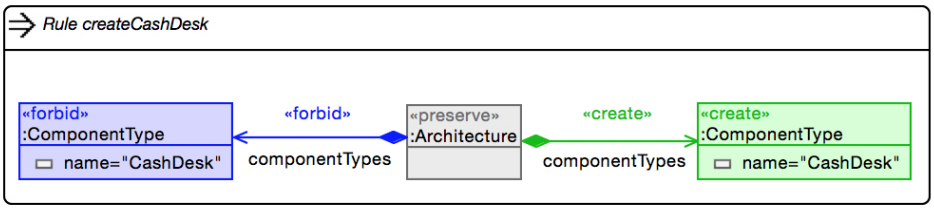


Figure 2.1: A simple Henshin model transformation rule

based on transformation definition files, Henshin also provides an API for executing model transformation definitions. In this thesis Henshin is used in the prototype, for transformations between different types of architecture models.

2.3.3 Triple Graph Grammars with HenshinTGG

Triple Graph Grammars (TGG) [Sch94, SK08] describe the simultaneous, context-sensitive production of three graphs. The three graphs are usually called source, correspondence, and target graph. The transformation rules of TGGs—so-called triple rules—describe a relationship between a source graph and a target graph, declared by interconnections via a correspondence graph. Forward and backward translation rules can be derived from triple rules. These derived rules define how a target graph can be created based on an existing source graph (forward rules) or vice versa (backward rules). Correspondence rules can be derived from the triple rules, that create a valid correspondence graph "between" the source and the target graph. TGGs are useful for creating bidirectional mappings between two representations of information.

HenshinTGG[3] [Lai13, Chapter 4] is a tool for defining TGGs and executing TGG transformations. It is technically based on Henshin, and operates on Ecore models. Triple rules are defined with Ecore-based meta models in HenshinTGG.

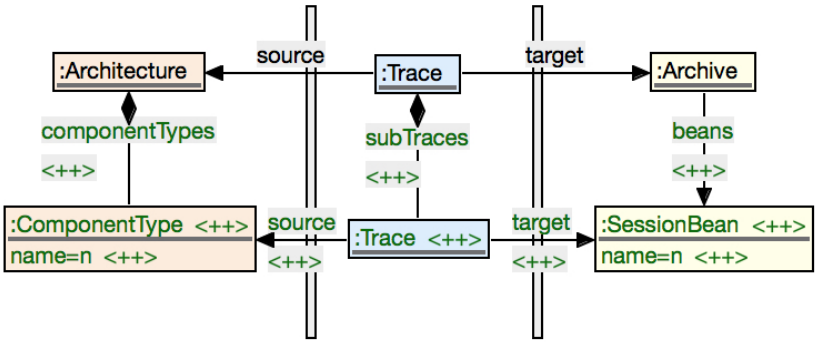


Figure 2.2: A simple HenshinTGG triple rule

Figure 2.2 shows a simple example of a triple rule created with HenshinTGG. The source graph of the triple rule (left side of the figure) describes an architecture that contains named

[3]HenshinTGG – `http://de-tu-berlin-tfs.github.io/Henshin-Editor/`

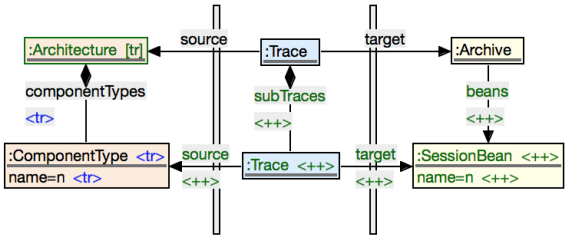


Figure 2.3: A forward rule derived from the simple rule in Figure 2.2

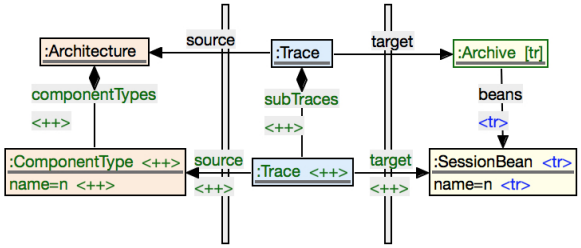


Figure 2.4: A backward rule derived from the simple rule in Figure 2.2

component types. The target graph (right side of the figure) describes archives that contain named session beans. The correspondence graph (center of the figure) defines *Trace* elements, which can have source and target nodes of any type, and other trace elements as subtraces. The rule describes that a *ComponentType* is related to a *SessionBean*. The name of the component type corresponds to the name attribute od the session bean. The rule relies on another rule, that relates an architecture to an archive.

From this triple rule, forward, backward, and correspondence rules can be derived. The Figures 2.3 and 2.4 show forward and backward rules derived from the triple rule in Figure 2.2. The forward rule defines that an *Architecture* element is searched for in the source graph that is related to an *Archive* via a *Trace* element as a basis. The architecture element must already be translated by another rule, hence the marker `[tr]`. For all instances of this structure, a match exists when the architecture has a reference to a *ComponentType* element, that has not been translated yet. These component types, and the corresponding *componentTypes* references, will be subject to translation. This is declared by the marker `<tr>`. The translation will create a new *Trace* and a new *SessionBean* element, and the corresponding references, as denoted with the marker `<++>`. The session bean's *name* attribute will have the same value as the corresponding name attribute of the component type. The backwards translation works analogously, by creating a new component type for each session bean. In this thesis HenshinTGG is used in the prototype, for transformations between different, model-based architecture languages.

2.3.4 Eclipse Java Development Toolkit

The Eclipse platform[4] provides an integrated development environment for a variety of languages. Eclipse plugins for Java development are concentrated in the Eclipse Java Development Toolkit (JDT)[5]. Most notable, JDT provides features for developing, building, and debugging Java programs.

The prototype developed in the context of this thesis is implemented based on the Eclipse platform. The API of JDT is used in the prototype to parse and change program code. This is used to create or change Java program code so that it contains program code structures which represent architecture model information, and to read program code to extract architecture models from it.

[4]Eclipse – `http://www.eclipse.org/`
[5]Eclipse Java Development Toolkit – `https://www.eclipse.org/jdt/`

2.3.5 Java Enterprise Edition

The Java Enterprise Edition (JEE) [Ora13b] is an umbrella standard for server side programs developed with the Java programming language. It comprises many APIs, most notable for dependency injection, transactional server side components, web services, object-relational mapping, and web-based user interfaces, In the context of this thesis, Context and Dependency Injection, Enterprise JavaBeans, and JavaServer Faces are used in case studies in the evaluation.

Applications using JEE as a platform are programmed against a subset of the JEE APIs, and deployed on JEE compliant application servers, such as the Red Hat's WildFly[6], or Oracle's GlassFish Server[7] for execution.

Context and Dependency Injection

Context and Dependency Injection (CDI)[8] [JSR17] is a technique to implement components as building blocks of software architectures in object-oriented languages. It manages the lifecycle of objects and their context, and references between these objects using a dependency injection mechanism. Java types managed by CDI are called *beans*. This term is overloaded multiple times within the JEE. Therefore in the remainder of this thesis this kind of beans is called *CDI beans*. Listing 2.1 gives a simple example of program code using CDI. It shows a simple CDI bean `TrivialBank` that provides two operations, and a simple CDI bean `CashDesk`, which uses the bank. The trivial bank type and the cash desk type are annotated as `RequestScoped`, which means that for each request up to one instance of these types are created. The request is the *context* of the bean instances. When the cash desk type is instantiated in the same context (during the same request), CDI injects the bank instance into the field after constructing the cash desk object.

```
@javax.enterprise.context.RequestScoped                                    1
public class TrivialBankServer {                                          2
    public boolean validateCard(...){...}                                  3
    public void debitCard(...){...}                                        4
}                                                                          5
                                                                           6
@javax.enterprise.context.RequestScoped                                    7
public class CashDesk {                                                    8
  @javax.inject.Inject TrivialBankServer bank;                             9
                                                                           10
  public void executePayment(){                                            11
    if(bank.validateCard(...))                                             12
      bank.debitCard(...);                                                 13
  }                                                                        14
}                                                                          15
```

Listing 2.1: Two interconnected CDI beans

Many types of properties can be used to determine which object should be integrated, both statically during development time or dynamically at run time. These features are not in the focus of this thesis. Within this thesis, the injection of CDI bean instances within their specific

[6]WildFly Application Server – `http://wilfdfly.org`
[7]GlassFish Application Server – `https://javaee.github.io/glassfish/`
[8]Context and Dependency Injection – `http://www.cdi-spec.org`

contexts is a feature used to identify components in JEE program code in a case study of the evaluation.

Enterprise JavaBeans

Enterprise JavaBeans (EJB) [EJB13] manages the lifecycle of objects of specific Java types and references between them, typically in the context of server-side programming. As such, it has common goals with CDI. Types managed by EJB are considered components in JEE compliant application servers. Among others, EJB provides mechanisms for transactions, component naming and discovery, security, and remote access, making it broadly used for server-side business applications.

Types managed by EJB are called *beans*, just like CDI beans. To avoid confusion, in remainder of this thesis this kind of beans is called *EJB beans*. Listing 2.2 shows the types of the CDI example as EJB beans. The `TrivialBankServer` and `CashDesk` are now annotated as `Stateless`, meaning that a pool of instances of these types may exist and each request gets one pseudo-random instance out of the pool. The cash desks references the bank server via a field. When an instance of the cash desk is used, an instance of the bank server is injected to the field. In contrast to the CDI example, the operation `executePayment()` is now executed within the context of a transaction. I.e. the results of the operation are only fixed, when the transaction completes successfully.

```
@javax.ejb.Stateless                                                    1
public class TrivialBankServer {                                       2
    public boolean validateCard(...){...}                               3
    public void debitCard(...){...}                                     4
}                                                                       5
                                                                        6
@javax.ejb.Stateless                                                    7
public class CashDesk {                                                 8
  @javax.ejb.EJB TrivialBankServer bank;                                9
                                                                        10
  // Executed within a transaction now                                  11
  public void executePayment(){                                         12
    if(bank.validateCard(...))                                          13
      bank.debitCard(...);                                              14
  }                                                                     15
}                                                                       16
```

Listing 2.2: Two interconnected EJB beans

This kind of beans is called *Session Beans*. Besides these beans, EJB also defines *Entity Beans*, i.e. data types for object-relational mapping and persistence, and *Message Driven Beans*, i.e. components that trigger and react upon events. Within this thesis, EJB Session Beans and their interconnections are used in a case study of the evaluation as components.

JavaServer Faces

JavaServer Faces (JSF) [Ora13a] is a standard for defining server-side web-based UI components in Java. In JSF web pages are defined using HTML, with additional tags for dynamic content. A relation to Java types is created using the *Expression Language (EL)*. Types managed by JSF are called beans. To avoid confusion, in the remainder of this thesis this kind of bean is called

JSF beans. Listing 2.3 gives an example of a UI defined with JSF. The listing shows an HTML web page with additional tags from JSF for dynamically building a table on each request. The data of the table is referenced in the parameter `value` of the tag `h:dataTable` using the EL. A *JSF bean* with the name `cashDeskUI` must contain a collection of elements `scannedItems`. Listing 2.4 shows the JSF bean `CashDeskUI`, which is mapped to the name `cashDeskUI` in the EL. It is annotated to be `ViewScoped`, meaning that it is a JSF bean, which is instantiated when the corresponding web page is *visited*, and exists as long as the site is *viewed*. The instance survives postbacks, i.e. when the user submits a form on the web page that has the same page as result, and also asynchronous requests. The operation `getScannedItems()` maps to the EL expression `scannedItems`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:h="http://xmlns.jcp.org/jsf/html" xmlns:jsf="http://xmlns.jcp.org/jsf">
<head>
    <title>CashDesk Web Interface</title>
</head>
<body>
    <form>
        <h:dataTable id="scannedItems" value="#{cashDeskUI.scannedItems}" var="item">
            <h:column>
                <f:facet name="header">Item Name</f:facet>
                <a jsf:outcome="itemDetails" title="Show #{item.name}">
                    #{item.name}
                    <f:param name="id" value="#{item.name}"/>
                </a>
            </h:column>
        </h:dataTable>
    </form>
</body>
</html>
```

Listing 2.3: A UI definition in JSF

```
@javax.faces.view.ViewScoped
public class CashDeskUI {
  private List<Item> scannedItems;

  public List<Item> getScannedItems(){ return scannedItems; }

  public void setScannedItems(){ return scannedItems; }
}

public class Item {
  private String name

  public Item(String name){ this.name = name; }

  public String getName(){ return name; }
}
```

Listing 2.4: A bean backing the UI definition of Listing 2.3 and its data type

JSF provides many features for building UIs. It can also be interwoven with CDI annotations

for creating interconnected JSF beans. Within this thesis, JSF beans and their interconnections with CDI are used in a case study of the evaluation as components.

2.3.6 Unified Modeling Language

The Unified Modeling Language (UML) [Obj15] is a generic graphical language that focuses on general software engineering tasks, such as requirements engineering, structural and behavioural object-oriented design, concurrency design, and deployment. It is widely used in software engineering in practice [LMWK14, SGT10].

The UML specification defines multiple types of diagrams, generally divided into *behaviour* and *structure* diagrams. Examples of UML behaviour diagrams are state machine diagrams that define a specific type of state machine, or sequence diagrams, which define scenarios or generic interactions between multiple run time elements. Examples for structure diagrams are the very common class diagrams, which can be used to specify interconnected classes in an object-oriented sense, or object diagrams that can be used to specify interconnected objects at run time.

In the context of software architecture, the component diagrams and composite structure diagrams are most notable. Component diagrams can be used to define components which are interconnected via interfaces. The components in that type of diagram are considered black boxes. Composite structure diagrams define the internals of composite elements, such as components. The combination of these two types of diagrams can be effectively used to describe architectural structures. In this thesis, the UML is used in case studies for evaluation. In these case studies, composite structure diagrams are derived from program code.

2.3.7 Palladio Component Model

The Palladio Component Model (PCM) [BKR09] is a component modelling language. PCM can be used for developing and simulating component-based software architectures, and to analyse the expected qualities based on their architecture specification. It is accompanied by the Palladio Simulator[9] as an IDE and simulator for quality properties [RBH+16].

Architecture specifications with Palladio are composed of five types of models: The *Repository* describes all components and interfaces of the system. The *Service Effect Specifications* (SEFF) describe abstract behaviour, including e.g. performance information or calls to required interfaces. The *System* is the high level run time view upon the defined software system. In this model components from the repository are instantiated. Analogously to the system model, composite components in the repository are refined in composite structure views, where they can instantiate other components from the repository, and interconnect the component instances via their interfaces. The *Resource Environment* model is used to describe the hardware nodes and network between them, on which a system should be executed. The instances in the system model are mapped to hardware nodes in an *Allocation* model. Finally, a *Usage* model is used to define a workload upon the system to analyse the system's qualities.

Within this thesis, the PCM is used in a case study for evaluation. In that case study, a repository and a system model are derived from program code.

[9] Palladio Simulator – `http://palladio-simulator.com`

3 Related Work

This chapter examines the current state of research regarding the relationship between program code and models thereof. Related work in adjacent fields is also considered.

3.1 Model-Code Co-Evolution

Langhammer [Lan17] and more abstract Langhammer and Krogmann [LK15] describe an approach for the co-evolution of Palladio architecture models and Java program code, including architectural structure and abstract behaviour in terms of Palladio's Service Effect Specifications (SEFF). Langhammer describes preservation rules, which preserve a consistency relationship between the architecture model and the program code during changes in either side. Change operations include the insertions, removal, and replacement of attributes, references, and objects in Ecore models. Arbitrary code within methods is preserved during model-to-code change propagation. The approach is semi-automated, meaning that in cases where full automation is not possible, a developer is asked to describe how consistency can be preserved. This approach creates a specific mapping between the specific ADL and the code. While the approach in general should be usable with multiple implementation or specification languages and can handle language evolution, it does not consider the differences between architecture implementation and specification languages.

Haitzer et al. [HNZ17] developed a method for software architecture and code co-evolution, that includes a mapping between program code elements and architecture model elements, and formalized evolution activities upon the architecture with translations into program code representations. They use a domain specific language to represent architectures. The mapping between program code elements and architecture model elements is limited to the identification of specific packages, classes, and interfaces in the program code, including sub- and supertypes, interface realizations, and dependencies. Changes in the architecture are first declared in the domain specific language and then implemented by developers in the program code. This approach does not include code generation, but solely checks for violations of architectural prescriptions.

Kapto et al. [KEKT16] describe a model-code co-evolution method that detects the application of architectural tactics [BBK03, p. 6] upon program code. They assume the existence of a mapping between elementary actions on program code, such as adding or removing a package, to architectural tactics. They present a language for describing architecture tactics, and detect elementary actions on program code by comparing its versions. The approach does not allow for changing code based on model changes.

Rocco et al. [RRIP14] explicitly describe language evolution as aspect of model-code co-evolution. When a system is modelled using meta modelled representations and corresponding code is generated, a challenge arises when the meta model is subject to evolution. Such changes can break the code generators. This is a case of model-code co-evolution: the meta model can be regarded as model and the code generator can be regarded as code in the context of model-code co-evolution. The authors propose a co-evolution approach where model changes are propagated

via well-defined transformations, which operate on the code and take the model difference as input. This approach can be used to handle architecture language evolution regarding model editors, but not regarding the code that implements a system's architecture.

Song et al. [SHC+11] bridge the gap between architectural descriptions and the actual system at run time. They assume the existence of two meta models, one that represents the architectural description at run time, and one that represents the actual system at run time. They use bidirectional model transformations to create causal connections between the two models. This allows for monitoring, and reconfiguration of the actual system in the syntax of the architecture model. The implementation is not considered in this approach.

3.2 Synchronization of Models and Synchronization of Models and Code

This section presents the current state of research regarding the synchronization of models with overlapping semantics and the synchronization of program code and corresponding models. Such synchronization is an inherent challenge for all approaches where multiple views upon a shared model exist. Modelling with multiple views has been subject to multiple approaches, some of them broadly accepted in the software engineering, such as the 4+1 view model of architecture [Kru95] or the architecture modelling standard ISO/IEC/IEEE 42010 [ISO11].

3.2.1 Consistency Management

Consistency management [FHK+15] (sometimes also referred to as *inconsistency management*) assumes that two views upon a shared body of information overlap. When one view is changed in the overlapping part, these changes should be propagated to the other view. Consistency management deals with methods and tools to reestablish synchronization.

The specification of trace links (also referred to as *traceability links*) can be seen as the most common technique for consistency management [WP10]. Change impact analysis is a method for measuring the impact that a change in an artefact has upon adjacent artefacts. Change impact analysis can be used to identify traces between artefacts or it can use trace links for measurement. Winkler and von Pilgrim [WP10] surveyed approaches for specifying and maintaining trace links in requirements engineering and model-driven development. They found that traceability has been subject to research since 1978 [Pie78].

Schenkhuizen et al. [SvdWJC16, Sch16] describe the Concern-Driven Inconsistency Management (CDIM) method for consistency management which can handle incomplete, informal, and heterogeneous software architecture models. The method is based on the inconsistency management process of Spandoudakis and Zisman [SZ01] and Nuseibeh et al. [NER01]. The method is at the moment not backed by a corresponding tool and therefore requires manual effort.

Multiple views upon models are a major concern in the systems engineering domain, where the description of systems comprise hardware, software, and often social concerns. The multi-disciplinary of this domain fosters inconsistencies between views. Feldmann et al. [FWKVH16] describe a consistency management approach for automated production systems. They use an explicit consistency model containing trace links between elements of the foundational models.

Consistency management is also used in further aspects. E.g. Karagiannis et al. [KBB16] describe a method for consistency management for enterprise modelling, where multiple inter-

dependent views – technical and social – exist upon a shared body of information. For managing the consistency, they employ a common ontology for the views, upon which they reason for consistency with queries. Ruhroth et al. [RGB+14] use atomic change operations on models and corresponding operations on code to keep code consistent with certain security models.

Consistency management approaches do not consider the gap between architecture specification and implementation languages.

3.2.2 Change Impact Analysis

Lehnert [Leh11a, Leh11b] developed a taxonomy for change impact analysis [BA96] approaches which he used to classify existing approaches. He concludes that most approaches to not span the whole software development process. One of their criteria is the scope of the analysis. Approaches are classified whether they analyse the code, architecture models, requirement models, miscellaneous artefacts, or a combination. Those approaches that analyse a combination of code and architecture models are of relevance to the Explicitly Integrated Architecture approach. Out of 160 approaches, Lehnert identified 19 that analyse a combination of scopes. Eight analyse a combination of code and architecture models. Three analyse a combination of code, architecture models, and requirement models.

Hammad et al. [HCM11] present an approach that identifies modelled elements that are impacted by a change in the underlying code. They only consider UML class diagrams. Changes in the architecture are not traced to their impact in the code.

Sharafat and Tahvildari [ST07, ST08] also analyse the impact of changes between Java program code and corresponding UML class diagrams. Their approach does not bridge the semantic gap between object-oriented structures in general purpose programming languages and higher architectural concepts such as components.

Bohner [Boh02b, Boh02a] and later Bohner and Gracanin [BG03] describe impact analysis between architecture models and code for systems composed of existing components and middleware. Their analysis is based on explicitly declared dependency graphs between architecture artefacts. Their focus is on adding middleware concerns to existing analysis technique. They do not describe how to link models and code.

Kim et al. [KKK10] focus on change impact analysis for software product lines. They link architecture models with code by using explicit mapping rules. With these mapping rules, model elements are mapped to program code directories or files. More fine grained code elements are not considered.

Hassan et al. [HDB10] describe an impact analysis approach between architecture models and program code. They employ two intermediate languages, one called Architectural Software Components Model (ASCM) to represent architectures independently from their original specification, and one called Software Component Structural Model (SCSM), for representing program code in the terms of their change impact approach. A projection is created between the ASCM and the SCSM, which ultimately describes trace links between the architecture model and the program code. This concept can also be used to handle evolving and emerging languages.

In addition to code and design, Ibrahim et al. [IIMD05, II05, IIM06] include requirement models and test cases in their impact analysis. Trace links in these approaches are gathered using three techniques: *explicit links* are created explicitly, *name tracing* creates links based on the naming of elements, *domain knowledge and concept location* uses the domain knowledge of the impact analysis users. Models or code are not generated.

Rostami et al. [RSHR15] describe the Karlsruhe Architecture Maintenance Prediction (KAMP) approach for change impact analysis. KAMP allows for a change impact analysis in architectures described with the Palladio Component Model (PCM) [BKR09]. In KAMP a relation can be established between program code and the architectural description by annotating modelled components with context information, including the location of program code files or configuration files. Upon modelling a change request, the KAMP approach can then show which files have to be changed, including metrics such as accumulated lines of code of all affected files.

3.2.3 Model-Code Synchronization

Völter [Voe10] describes a projectional Integrated Development Environment (IDE) that can represent models, including architectural structure and behaviour models, within the program code. The visual projection is created on the fly, based on the abstract syntax tree (AST) of the underlying code. Changes in the model are translated into changes in the AST. The Explicitly Integrated Architecture approach could be used in this context to create such projections of architecture models. The approach does not handle language evolution.

Already in 1995 Murphy et al. [MNS95] presented an approach for bridging the gap between program code elements and higher-level software models. In their approach a mapping is created between higher-level model elements and program code elements. A so-called *reflexion model* is created as a view, that compares the prescriptive higher-level model with the descriptive program code, by showing the convergence and divergence between program code and model elements and the absence of program code elements compared to the prescriptive higher-level model based on the mapping. Murphy et al. use a mapping between program code files and model elements. The general ideas behind this approach is close to the Explicitly Integrated Architecture approach. In both approaches a semantic gap between program code elements and higher-level models has been identified and should be bridged with a mapping. Instead of program code files, the Explicitly Integrated Architecture approach uses a formal model of the program code based on the elements provided by the underlying programming language as basis for the mapping. That formal model is mapped to meta model elements of an architecture languages with transformations. The Explicitly Integrated Architecture approach is therefore more expressive. Language evolution is not handled by Murphy et al.

ReflexML of Adersberger and Philippsen [AP11] is a mapping of UML component diagrams to program code artefacts, enriched with a set of consistency checks between the model and the code. They defined a UML profile with stereotypes for referencing type or interface declarations that implement a component or an architectural interface. They use the ideas of aspect-oriented programming for pointcut declarations to create that mapping using e.g. wildcards or by defining supertypes. The predefined conformance checks can then be used to validate whether the code conforms to the prescriptive architecture model.

Code Generation

Code generation [SLS18] has the aim to generate program code from a model specification. The idea is often, that the generation happens once. The resulting program code can then be adapted and extended to the specific need of the project.

As Syriani et al. state, “Code generation has been around since the 1950s, taking its origin in early compilers” [SLS18]. In this thesis, code generation is seen as deterministically deriving

program code from a meta modelled model. Therefore typically templates are used as an input for code generation. Relevant as related work are code generators for higher-level models. E.g. Shluga et al. [SISV16] describe a code generator for state machines. Das et al. [DGJ+16] generate code for embedded systems based on UML for Real-Time (UML-RT) [Sel98]. Their generated code includes monitoring aspects. Gessenharter and Rauscher [GR11] generate code for activity diagrams.

Further code generators exist for architectural aspects. Cavalcante et al. [COB14] generate program code in the Go programming language [GoP] based on architecture descriptions with the π-ADL [Oqu04], including components, connections, and abstract behaviour specifications. Lung et al. [LRSS10] generate code for architecture patterns in distributed systems, allowing for rapid architecture prototyping. Further approaches generate code based on arbitrary domain specific languages [EBM12, Pü16].

Approaches focusing on code generation transform models to code. When the model changes, the code should be generated again to reflect the changes. This imposes the challenge that changes in the generated code may be overwritten. Manual changes in the generated program code can break the synchronization with the originating model. When the specification language evolves, a new generator is required. A relation to existing code is not considered in this case by code generation approaches. When the implementation language evolves, an initial code base can be generated for the evolved language. Code added manually to the previously generated code base has to be manually copied and adapted.

Model-Driven Development and Model-Driven Architecture

In Model-Driven Software Development (MDSD) [SVC06] – also called Model-Driven Engineering (MDE) – domain specific models of parts of a program or whole programs are created with domain experts. These domain models are refined with detailed technical models that are not relevant to the domain, but to the platform that will run the software. Such models are the basis for automated code generation. The generated code has to be enriched with implementation details. Model-Driven Software Development is a broadly accepted and employed method [WHR14], especially in the domain of embedded systems [RAK15, LMT+14, ZT13, SV12, Cor10], but also in the domain of information systems [DMWW15]. Model-Driven Architecture (MDA) [Gro14] is a MDSD approach for software architecture. In MDA platform-independent models (PIM) are the domain models. Platform-definition models (PDM) are the basis for translating PIMs into platform-specific models (PSM). PSMs, or program code generated from them, can be run on their corresponding platform. MDSD concentrates on deriving program code from models. The specification (PIM, PDM, and PSM in MDA) and the program code are two interdependent views upon the architecture that are subject to evolution and maintenance independently. Changes in the specification can be taken over automatically in the implementation. When the architecture changes in the implementation, these changes cannot be automatically taken over in the specification. MDSD bridges the gap between the abstraction levels of the representations, but changes can only be carried over one way, from the abstract specification to the detailed program code. As MDSD and MDA are a special case of code generation, language evolution is not addressed by these approaches.

Model Extraction

The aim of model extraction methods is to create a model of a software based on code or execution traces. Such architecture model extraction methods are based on techniques for identifying components in program code. Birkmeier and Overhage made a survey on methods for identifying components in 2009 [BO09]. Many approaches for model extraction have been developed over time. E.g. Srinivasan et al. [SLY16] extract sequence diagrams from Java program code and execution traces. Sen and Mall [SM16] extract finite state machines of arbitrary Java programs using symbolic execution. Other approaches can be used to extract interaction models between the user and the program [Bow15], variability models from plugin descriptors [ACC+14], or platform independent models of web applications [RS13]. These approaches create models used for understanding or analysing the program. They usually aim to create lower-level models such as UML class diagrams or sequence diagrams, and therefore do not bridge the semantic gap between higher-level models and program code.

Architecture models are also subject to model extraction methods. Ducasse and Pollet published a taxonomy for software architecture reconstruction approaches and compared existing approaches with their taxonomy [DP09]. The following is a selection of software architecture reconstruction approaches.

SoMoX [BHT+10] is an architecture model extraction tool that uses metrics to identify components and their interconnection within program code. The extraction method is based on heuristics and can be tuned by the user to get the desired architecture description.

With the concept and tool *Archimetrix* [vDPB14], von Detten et al. extract component-based architecture models even under the circumstances of design deficiencies, using a semi-automatic approach. Archimetrix uses SoMoX for the detection of components based on metrics, before the method's user is guided to incrementally provide input on how to get the desired architecture description, thereby removing design deficiencies in the underlying program code.

Alshara et al. [AST+16] extract component models from program code. They detect clusters of classes in the program code and distinguish between internal and boundary classes. Internal classes are used internally by the cluster and have no direct connections the cluster's context. Boundary classes serve as entry points into the cluster.

Weinreich et al. [WMBK12] describe a method for extracting architecture models from software implemented with a Service-Oriented Architecture (SOA) [TMD09, pp. 443]. A model of the ADL *LISA* is built by parsing the program code, byte code of libraries, and component specifications of several component specification languages.

Chouambe et al. [CKK08] argue that a manual approach for reconstruction would be better than a fully automated approach. As a consequence, they present a semi-automatic method, which can be used to assist an architect in reconstructing the architecture from the program code, by suggesting possible components, based on metrics.

These methods for identifying components are usually based on heuristics and rely on generic assumptions about component-based design: most prominently they assume that a component is a cluster of artefacts with high cohesion, which is not always true in practice. Also, when the resulting architecture models should be the basis for analysis, one must be confident that the models are complete and correct to a degree that does not affect the analysis results. With heuristic methods or methods based on generic assumptions about component-based design this confidence cannot be ensured. It should be possible to tune the extraction based on technical, domain, and project knowledge. The Explicitly Integrated Architecture approach allows for encoding technical knowledge using well-defined transformations for specific architecture

implementation languages, and project knowledge when such transformations are used as a basis for defining project-specific transformations. Language evolution is not an issue for model extraction approaches in general. When the implementation or specification language of a system's architecture changes, a new extraction logic has to be implemented.

Model extraction methods in general can be only a part of the Explicitly Integrated Architecture approach, because changes in the model are not forwarded to changes in the underlying program code.

Program Comprehension

The field of program comprehension [Sie16] is concerned with helping developers to understand the software. A part of program comprehension is usually the extraction of models, such as architectural models, object-oriented models, but also e.g. metrics. Kozar et al. [KMC12] showed that program comprehension is easier with models of a domain-specific language than without.

With program comprehension as a goal, several model extraction approaches have been developed. Some examples are the following: Ebert et al. [EKRW02, EB10] describe *GUPRO*, a generic framework for program comprehension. GUPRO creates a graph as an abstract view upon the program code, which can be queried by GReQL [GRe01] queries to find relevant structures. Haiduc et al. [HAM10] create a textual summary of program by scanning the code for important artefacts. Abi-Anoun et al. [AGCK14] extract information about objects and call graphs between them out of the static program code.

As the goal of program comprehension approaches is the understanding, not the evolution of programs, they do not provide means to edit the program code in the model views. The Explicitly Integrated Architecture approach can be used in the context of program comprehension, as it is usable to extract architecture models of a program. As an extension, the model view can also be used to change the mapped code.

Roundtrip Engineering

Roundtrip engineering [Aß03, NNWZ00] is a method where two views of program code are maintained together: in a textual syntax and in a – usually graphical – model syntax. In this context the method offers a bijective projection between the textual and the model syntax. The models used in roundtrip engineering are close to the code structures, e.g. UML class diagrams or data models. The Explicitly Integrated Models approach can be seen as a case of roundtrip engineering with architecture models and program code.

3.2.4 Embedded Models

Balz [Bal11] describes an approach for representing models with well-defined code structures, called *Embedded Models*. Balz defines *Embedded Models* as a mapping between formal models and program code patterns in a general-purpose programming language. In this context the patterns are views upon model structures in the syntax of the programming language.

A major contribution of Balz' work is the formal mapping between higher-level models, such as state machines and process models, and structures in programming languages, which includes the definition of two layers of model semantics. One layer describes the semantics declared for a meta model. Every instance of that meta model uses these semantics. E.g. a transition in a state machine always first evaluates a guard, then executes an update, and at last changes the

current state of the state machine to the target state. These concerns are always executed in this order. For each meta model, an execution runtime is developed, which executes the semantics of the meta model and is parametrized with the specific model to execute. Balz employs well-defined interfaces within the program code patterns to be executed by these runtimes, e.g. a method *guard* to be called in a type declaration that represents a guard in a state machine.

The other layer describes the semantics of the specific modelled element. E.g. the update definition of a specific transition should have a specific effect on the application that is controlled by the state machine. In a bank account example, the balance of an account could be decreased. It could be expected in a traditional program, that such a change should be reflected in a database. To integrate with arbitrary program code, the Embedded Models approach defines interfaces for this kind of semantics. Balz uses an *actor* as representative for the context in which the state machine is embedded. The actor provides operations for the semantics that the state machine can execute on its context.

The Model Integration Concept (see Chapter 5) is conceptually based on the foundational ideas of the Embedded Models approach. It also uses program code patterns to represent model information. Balz defines two specific types of models for which embedded models can be used, state machines and process models. The Model Integration Concept generalizes this approach by building upon a definition of meta models (see Section 5.4.1). Any meta model that can be defined with these means can be used in the Model Integration Concept. With the definition of meta models as a basis, the Model Integration Concept can declare *integration mechanisms* (see Section 5.6) as general templates for program code patterns. In prior work we defined model representations in code and corresponding execution runtimes, including Interface Automata [MBG11a], Protocol Contracts [KKG14, KG14], State Machines [Kon14], or Palladio Allocation models [MK16]. These can be seen as applications of Embedded Models.

3.2.5 Model Execution

Model execution handles models as data for model execution engines. Popular executable modelling languages are Executable UML (xUML) [MB02], the graphical Foundational UML (fUML) [Gro16], and the textual Action Language for fUML (Alf) [Gro13], which all provide means to add formal, executable semantics to UML models. Oquendo et al. [OLB16] integrate Alf expressions into the ADL SysADL, a profile for SysML [Gro15a], to get executable architectures. Khan and Haider [KH14] present an ADL that combines ontologies, UML, and Colored Petri Nets to model architectural structure and behavior. Putschögl and Dorninger [PD10] add interaction information to UML activity diagrams to make them executable.

ArchJava [ACN02] adds the notion of components, connectors, and ports to Java program code. Composite components can interconnect their composed components with connectors via their ports. ArchJava adds (textual) architecture model information to Java programs. It is translated into byte code for the Java virtual machine by the ArchJava parser. The execution engine in the sense of model execution is the compiled byte code.

Executable models are based on model runtimes, which implement the semantics for a meta model. These runtimes are comparable to runtimes for program code structures in the Model Integration Concept. Instead of using a model definition as input (as a runtime for executable models would do), such a runtime takes program code structures as input, which represent model elements, and executes the semantics as declared by a specification of semantics for the meta model. Model execution approaches in general do not address language evolution. While it is possible to create model transformations for translating models into evolved or emerging

languages, this is not within the scope of the model execution research area.

The Model Integration Concept fits well into the field of model execution. This has been shown in prior work, where we defined model representations in code and corresponding execution runtimes for different types of models. The resulting program code is executable in combination with the corresponding execution runtime and can therefore be regarded as executable models.

Research Area	Reference	R1	R2	R3	R4	R5
		Bridges Gap	Language Differences	Single Source	Bidirectional	Language Evolution
Model-Code Co-Evolution	[LK15]	✓	-	✓	✓	✓
	[HNZ17]	✓	-	-	-	-
	[KEKT16]	✓	-	✓	-	-
Change Impact Analysis	[Boh02b, Boh02a, BG03]	✓	-	-	-	-
	[HDB10]	✓	-	-	-	✓
	[IIMD05, II05, IIM06]	✓	-	-	-	-
	[RSHR15]	✓	-	✓	-	-
Model-Code Synchronization	[Voe10]	✓	-	✓	✓	-
	[MNS95]	✓	-	-	-	-
	[AP11]	✓	-	-	-	-
Code Generation	[SISV16]	✓	-	✓	-	-
	[DGJ+16]	✓	-	✓	-	-
	[GR11]	✓	-	✓	-	-
	[COB14]	✓	-	✓	-	-
	[LRSS10]	✓	-	✓	-	-
	[EBM12]	✓	-	✓	-	-
	[Pü16]	✓	-	✓	-	-
MDSD and MDA	[SVC06]	✓	-	✓	-	-
	[Gro14]	✓	-	✓	-	-
Model Extraction	[BHT+10]	✓	-	✓	-	-
	[vDPB14]	✓	-	✓	-	-
	[AST+16]	✓	-	✓	-	-
	[WMBK12]	✓	-	✓	-	-
	[CKK08]	✓	-	✓	-	-
Embedded Models	[Bal11]	✓	-	✓	✓	-
	[Kon14]	✓	-	✓	✓	-
	[MK16]	✓	-	✓	✓	-

Table 3.1: Overview of the Evaluation of Related Work

3.2.6 Summary

Section 1.6 defines the requirements towards an approach for bridging the gap between architecture specifications and architecture implementations. In this section, the current state-of-

research was evaluated regarding these requirements. The results are summarized in Table 3.1. The table only shows related work that is bridging the gap in general. Approaches that do not bridge the gap are not considered here.

3.3 Adjacent Research Areas

In the following sections, adjacent research areas are inspected.

3.3.1 Model-Based Migration

When legacy software is to be migrated to modern platforms or languages, a common practice is the Model-Based Migration (MBM). The underlying challenge of MBM is the evolution or emergence of implementation languages. Several experience reports for MBM have been presented, especially in the context of the workshop series *Software-Reengineering & Evolution (WSRE)*[1]. These methods usually follow the same principles. First a meta model for a domain specific language is developed that can represent the semantics of the legacy system. Then a meta model is developed for the target platform and transformations between these meta model are created. This is usually done in an iterative process, until most of the code can be translated automatically. Many examples of such migrations exist [RCM14, AB14, BT12, BCJM10].

The Explicitly Integrated Architecture approach can be used in this context by following the stated principles, but is limited to well-defined code structures. The legacy system can be translated into a model of the Intermediate Architecture Description Language and from there to a more recent platform. This only translates code, which is part of mappings between the model and code representations, and does not take e.g. detailed behaviour into account. Such a migration is executed in a case study in Section 10.4.

3.3.2 Architecture Interchange Languages

It might be necessary to migrate a given software architecture specification model into another language. Possible reasons include that analyses, that should be executed, are not possible in the used specification language, or the language evolved, and a newer version should be used. In these cases an architecture interchange language can be used to decrease the effort for translation. Architecture interchange languages provide means to describe arbitrary properties that can be interpreted by translations into other languages.

ACME [GMW97] is an early ADL that has specifically been developed as an architecture interchange language. ACME has language elements to model *systems*, *components*, and *connectors*. *Ports* are interfaces of components to their context. Connectors interconnect components by binding their ports to *roles* within the connector. All elements in an ACME description can be enriched with *attachments*. Attachments contain arbitrary content, which is ignored by the ACME tools. Translations from or to other ADLs can generate or interpret these structures and attachments.

KLAPER (Kernel LAnguage for PErformance and Reliability analysis) [GMRS08, GMS05] is an intermediate language for the performance and reliability analysis of component-based systems. The main idea of KLAPER is that component-based systems are designed using languages that are well suited for software design, but that for system analysis other languages

[1] WSRE – `https://fg-sre.gi.de/archiv/wsr.html`

are to be preferred. KLAPER architectures comprise software or hardware *resources* that offer *services*. Such resources are typed and may have attributes, including performance attributes. KLAPER offers typed attributes for exchanging architecture information.

A more general approach for exchanging models is the XML Model Interchange format (XMI) [Gro15b]. XMI defines an XML-based format for exchanging model information for meta models based on MOF as common meta meta model. For architecture languages based on MOF, XMI can be used as in interchange format.

When an architecture is expressed in one of many architecture implementation languages and should be translated into one of many architecture specification languages, then an architecture interchange language can reduce the number of translations necessary to define. The Explicitly Integrated Architecture approach includes the Intermediate Architecture Description Language in the role of an architecture interchange language in Chapter 6.

3.3.3 Modular Architecture Languages

A meta model can be seen as an interface between its models, and all tools working with these models. Most architecture specification languages are based on a single grammar or a single meta model. Malavolta identifies challenges imposed by this monolithic structure [Mal10]. These languages are not *extensible*, *customizable*, and do not provide suitable means to manage *multiple views*. To overcome these challenges, Malavolta describes byADL (*build your ADL*) [DRMM+10], *DUALLY* [MMPT10], and *MEGAF* [HMMP10]. *byADL* is a framework to incrementally compose ADLs using composition rules. *DUALLY* is a transformation framework for transforming architecture descriptions into the syntax of other languages. *MEGAF* is a multi-view framework to use with byADL.

Strittmatter et al. [SRHR15] describe the modularization approach of the Palladio Component Model (PCM) [BKR09]. They use a language kernel with multiple layers of interdependent modules to keep the language extensible, customizable, and to provide multiple views. The Intermediate Architecture Description Language presented in this thesis (Chapter 6) is a modular architecture language.

3.4 Summary and Conclusion

Some approaches relate implementation artefacts to architecture elements, but for many of those the support is limited to directories or program code files. Only five of the remaining approaches have bidirectional translations between both concepts. Only two of the remaining approaches explicitly address language evolution. None of them is prepared for differences between the architecture languages.

The evaluation of the related work shows that currently no general approach exists for bridging the gap between architecture specifications and architecture implementations, that allows for bidirectional translations between architecture implementation and specification languages while considering the differing focus of these languages. The related work does not structurally and efficiently handle the evolution and emergence of architecture languages. This thesis' objective (see Section 1.6) thereby justified.

Part II

Bridging the Gap between Software Architecture Specifications and Implementations

4 Proposed Solution

Chapter 3 shows that currently no general approach exists to bridge the gap between architecture implementation and architecture specification languages, that fulfills the requirements stated in Section 1.6. This chapter gives an overview of the solution proposed in this thesis to fill this gap. Using the proposed approach, architecture model information is integrated with program code. The code will contain sophisticated structures, which represent architecture meta model and model elements, and their properties, while respecting the requirements, that architecture implementation languages have towards the program code.

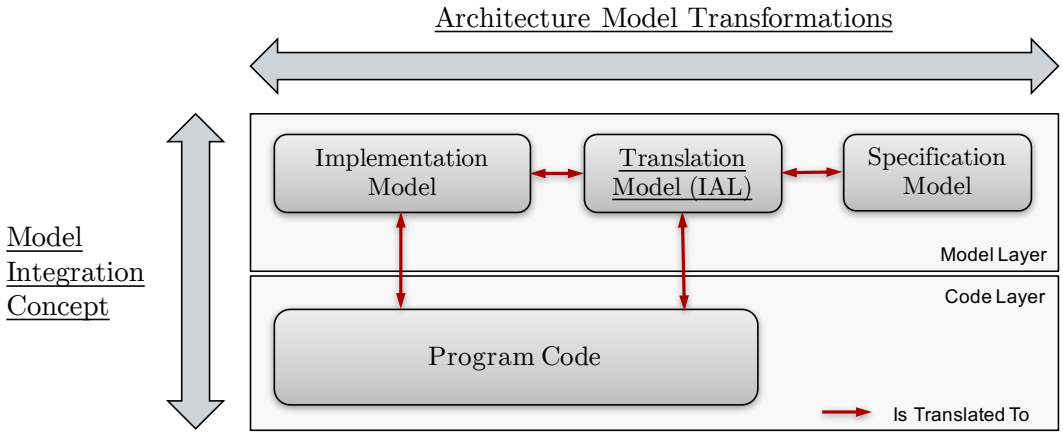


Figure 4.1: The parts of the proposed solution (underlined), the elements, and their interrelationships

Figure 4.1 gives an overview of the view types, that are subject to translation in the proposed solution, and their interrelationships. The figure will be subsequently described in the following chapters. The view types that are subject to translation are the following:

Program Code: the implementation of a software following the standards of an architecture implementation language

Implementation Model: an abstract model view upon the program code, that complies to an architecture implementation language

Translation Model: an intermediate model view for translating between an implementation model and a specification model

Specification Model: a specification of architectural concerns using an architecture specification language

Four research areas are part of the proposed solution:

1. A **Model Integration Concept** is used to integrate models and meta models with program code. It is used to create well-defined translations between program code structures, model elements, and meta model elements.

2. The proposed solution uses the translation model during the translation between architecture implementation and architecture specification models, for reducing the number of required translation definitions. The **Intermediate Architecture Description Language (IAL)** is defined to express translation models.

3. **Architecture Model Transformations** are used for the translation between models of different languages, and for transformations within models of IAL.

4. The **Explicitly Integrated Architecture Process** describes how these areas are used to achieve the objective.

For all these research areas, concepts and implementations are developed in this thesis. In the following sections, these research areas are briefly described to give an overview of the approach. The research areas are then described in detail in the following chapters. The idea of the presented approach is to ensure the consistency between the architecture views at development time. The consistency with the run time has to be subject to an execution runtime environment, and is not in the focus of this thesis.

4.1 Model Integration Concept

The Model Integration Concept describes how model information[1] is integrated with program code. In Figure 4.1 the concept provides vertical integration. It is used to integrate and extract architecture model information from an implementation model and the translation model with/from program code (see [KG12]). For doing so, within the Model Integration Concept it is defined how information of meta models and models thereof can be notated with program code structures. The code is statically analyzed for program code structures that identify implementation model elements, or adapted respectively when the model is changed. It therefore derives a new view with bidirectional mappings. The Model Integration Concept is described in detail in Chapter 5.

4.2 Intermediate Architecture Description Language

The Intermediate Architecture Description Language (IAL) is a translation model language. It is used to represent architecture information independently from the specification language that is used to describe the architecture and from the implementation model that is used to implement the architecture. It has the role to increase the interoperability of the proposed solution with different specification and implementation models, and to increase the evolvability of the approach. Chapter 6 describes the Intermediate Architecture Description Language. Specification and implementation languages have different kinds of information that they are able to describe. E.g. in contrast to specification languages, implementation languages often

[1] Models in the term of this approach are always based on meta models. Other models, such as mathematical functions etc. are not meant here.

cannot describe a deep component hierarchy. The IAL handles these differences using a profile concept similar to UML profiles [Obj15, Chapter 12.3]. The IAL is described in detail in Chapter 6.

4.3 Architecture Model Transformations

In Figure 4.1 the architecture model transformations provide the horizontal integration. Two kinds of transformations are used within the proposed solution: First, specification models are synchronized with implementation models via a translation model. Second, translation models may have to be transformed to match the different kinds of information required by the targeted specification or implementation model language. Both kinds of transformations are described in Chapter 7.

4.4 Explicitly Integrated Architecture Process

The three parts above build the conceptual foundation for a process for automatically integrating architecture model information with program code and extracting this information. Figure 4.2 shows a simple example of the Explicitly Integrated Architecture Process in action. In this example an EJB Session Bean `CashDesk` is added to an existing bean `BarcodeScanner`. The `CashDesk` is declared to be the parent of the `BarcodeScanner`.

(1) shows the program code for the bean `BarcodeScanner`.

(2) The implementation model is built by scanning the program code for well-defined structures based on the Model Integration Concept. In this example a type declaration with an attached annotation `Stateless` is identified. The name of the declared type is identified as the name of the bean.

(3) The implementation model is translated into a translation model, an instance of the IAL.

(4) The translation model is translated to a specification model. The specification model in the example is represented using a UML component diagram. In an evolutionary step, a parent component named *CashDesk* is added.

The changes are propagated to the code as follows:

At **(5)** the architecture specification model is translated into the translation model. A new *ComponentType* with the name *CashDesk* is created, with a stereotype that allows to add children to a component type.

(6) The translation model is translated into an implementation model. In this model the hierarchy cannot be represented, because the EJB specification does not define component hierarchies.

At **(7)** the program code is adapted corresponding to the changes in the implementation model. I.e. the type `CashDesk` is created.

(8) The architecture information that has no representation in the implementation model is translated into the code using the Model Integration Concept. In this example, the hierarchy is translated as a field in the Java type declaration `BarcodeScanner` with the annotation `EJB`. This is an annotation of the EJB framework, that specifies that an instance of the bean `BarcodeScanner` has to be injected. Additionally this field has the annotation `ChildTypes`, which marks the reference an instance of the *childTypes* reference. To remove the hierarchy the code could be translated to a model using the process. As an alternative, the respective code element could be removed.

It should be noted that the hierarchy could also have been created in the terms of the approach by simply adapting the code accordingly, because the models can be derived automatically. The Explicitly Integrated Architecture Process is described in detail in Chapter 8.

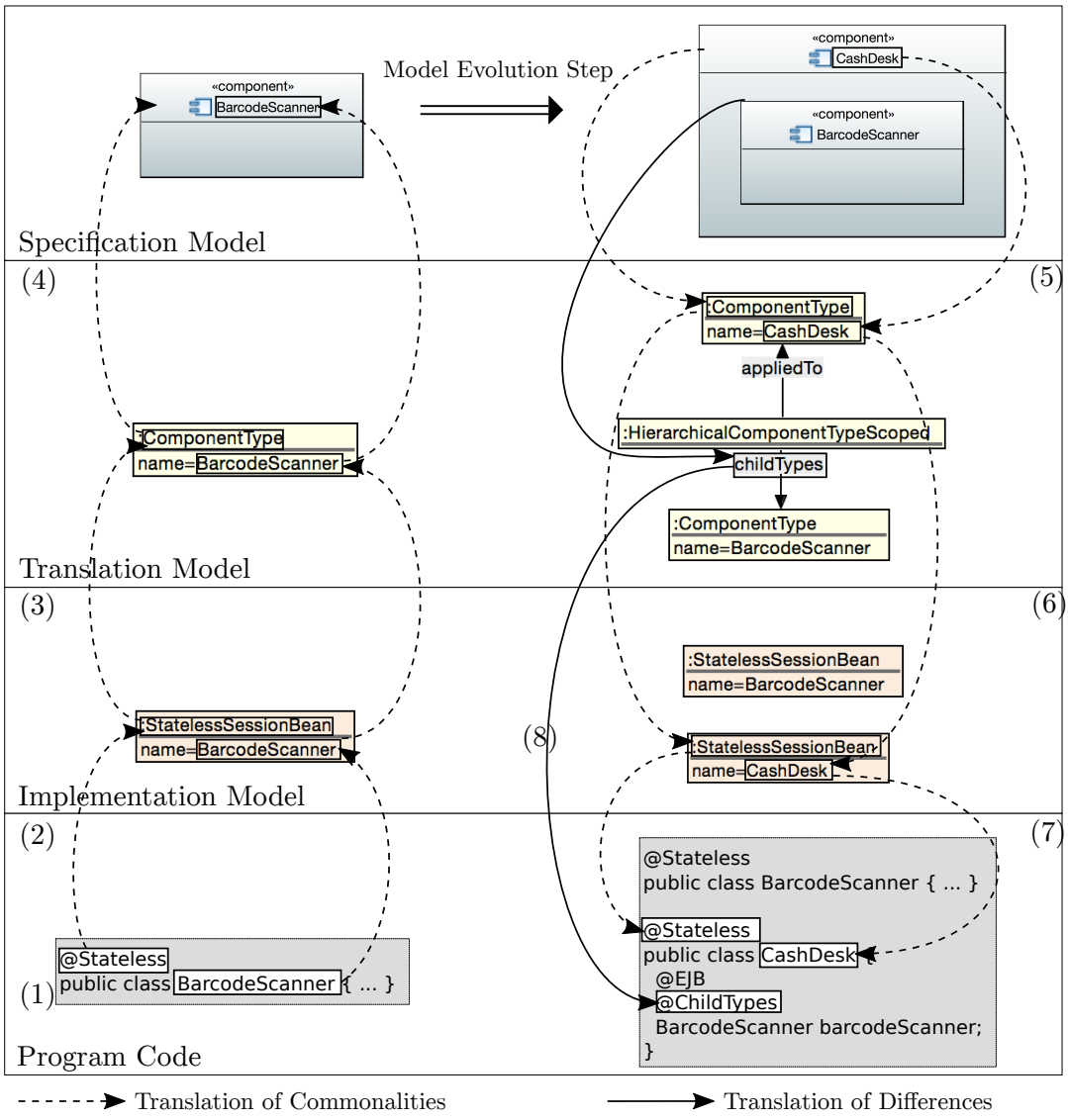


Figure 4.2: An Example of the Explicitly Integrated Architecture Process

This chapter gave an overview of the proposed solution and sketched the parts of the solution and their interconnection. In the following chapters, these parts are described in detail.

5 Model Integration Concept

The Model Integration Concept integrates architecture model information with program code. Figure 5.1 highlights its role in the proposed solution. We first state the objective of the concept and the foundational assumptions for its applicability in Section 5.1. Section 5.2 gives an example of *notations*—a formal mapping between models (or meta models) and program code structures. After an overview of the concept in Section 5.3, a formal definition of languages involved in the concept, their required elements and their interrelationships is provided in Section 5.4. This definition is the basis for the definition of *notations* of information in different languages, which is given in Section 5.5. Section 5.6 identifies patterns for notations, called *integration mechanisms*. At last, Section 5.7 describes the process how transformations for specific languages are developed.

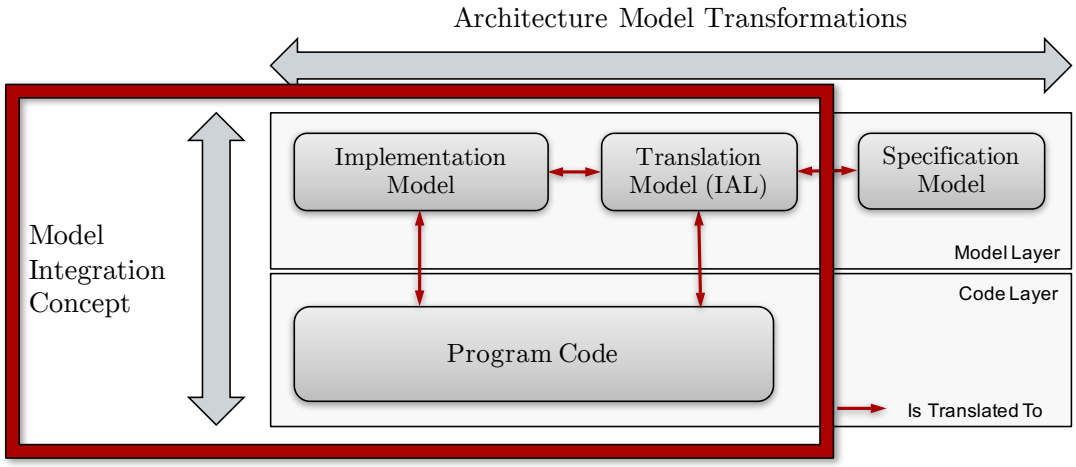


Figure 5.1: The Model Integration Concept highlighted in the overview of the proposed solution

5.1 Foundational Assumptions for Integrating Models with Program Code

The objective of the Model Integration Concept is to integrate model information with program code written in an object-oriented general purpose programming language. It assumes the existence of two languages. Both languages must be a described with meta models. One language must be an object-oriented general purpose programming language. This language is called the *programming language* in the context of this concept. The information modelled with this language is called *program code*. The other language is an arbitrary other language. The information modelled with this language is called *model*.

5.2 Example of an Integrated Model

Figure 5.2 shows a very simple example of an integrated architecture model.[1] The meta model of the modelling language on the upper left side of the figure comprises one class *ComponentType* with the attributes *name*, and *version*, both of the type *String*. The lower left side shows an instance of that meta model, a single object of the *ComponentType* class, with the name `BarcodeScanner`, and the version `1.0`. The right side shows their program code representation with Java as a programming language. The lower right side shows a translation of the object. The program code declares a type `BarcodeScanner`. An annotation `ComponentType` is attached to the type, with the annotation member `version` set to `1.0`. The upper right side shows the translation of the meta model element. The meta model class is represented with the declaration of an annotation with the name `ComponentType`. It has an annotation member `version`, of the type `String`. The declaration of the annotation `ComponentType` is shown in the upper right side.

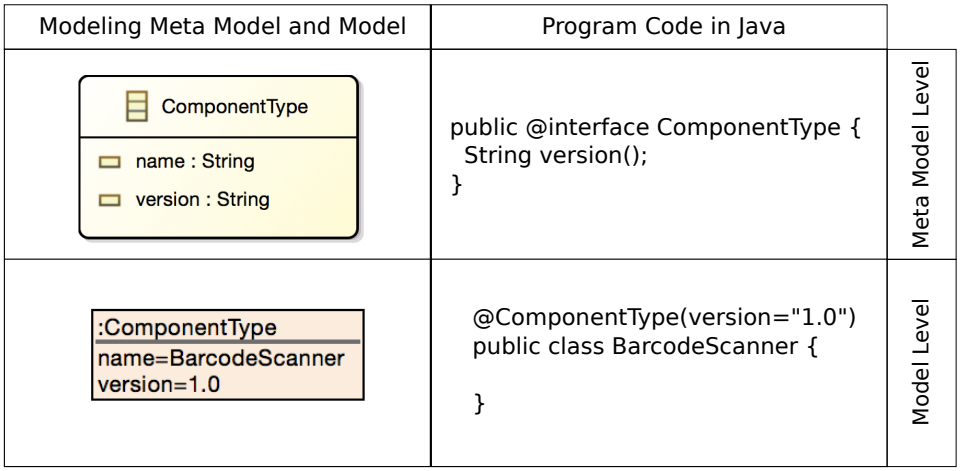


Figure 5.2: Example of an integrated model. The upper left side shows a meta model. The lower left side shows an instance of that meta model. The lower right side shows an program code representation of the model. The upper right side shows an program code representation of the meta model.

The relation shown in Figure 5.2 can be the basis for defining transformations between model elements and program code, or meta model elements and program code. Such well-defined translations have to ensure that the translation of a model element or a meta model element into program code as well as the interpretation of model structures from the program code are unambiguous. The relation between the meta model element and the annotation declaration in the example in Figure 5.2 is based on the equality of the name of the meta model element and the annotation declaration. A transformation based on this relation is executed once. As long as the meta model remains unchanged, the translated code structure can be reused as a library for programs to reference. The example also shows the relation of the program code

[1] The notation used for expressing meta model and model elements here and in following examples is related to UML. They do, however, not represent UML classes and objects.

and meta model view on the attribute *version*: The attribute is expressed as declaration of an annotation member. Its type is declared correspondingly.

The relation between the model element and the type declaration with the attached annotation in the example is based on the relationship between the type declaration and the attached annotation. The fact that the annotation is attached to the type defines that this type represents an instance of a *ComponentType* meta model element. The name of the type is declared to be equivalent with the value of the attribute *name*. The value of the annotation member `version` is equivalent to the value of the attribute in the corresponding model element.

The example shows that two categories of bidirectional translations are necessary, one between the meta model and program code, and one between the model and program code. The translations have to be set up so that the relationship between meta model elements and their model instances are consistently translated into program code. E.g. the type of the annotation member *version* enforces that the values entered can be used as values for the attribute.

The Model Integration Concept declares translations between the abstract syntax elements of models and code, and their respective language elements. The translations must preserve the semantics of the translated elements. When an execution runtime exists that interprets the annotation of the type declaration to create a component instance, then the semantics of the model element is declared in the execution runtime. The runtime could e.g. manage a life cycle for the component instance, and announce its existence in a component instance registry. These semantics must be reflected in the modelling language element that represents this code element.

An execution runtime could also call methods within the declared type to execute behaviour. The behaviour is not part of the meta model presented in Figure 5.2. It could be subject to another modelling language, that is also integrated in the type declaration, or not be modelled at all, in which case it would be considered an implementation detail in the context of the Model Integration Concept. Therefore the *body* of the type declaration is called an *entry point* for further details of the program.

5.3 Overview of the Parts of the Model Integration Concept

Figure 5.3 gives an overview of the elements in the Model Integration Concept and their interrelationships. The Model Integration Concept consists of the following elements.

- *Modelling Language Meta Models and Models:* Modelling language meta models define the abstract syntax of a modelling language. Models are instances of these meta models. In Figure 5.3 the modelling languages meta models and models are shown on the left hand side.
- *Programming Language Meta Models and Program Code:* Programming language meta models in the context of the Model Integration Concept define the abstract syntax of a programming language. Program code is an in an instance of such a meta model. In Figure 5.3 programming language meta models and program code are shown on the right hand side.
- *Modelling Language Meta Model Code Structures:* Modelling language meta model code structures are parts of a program, that are defined to represent meta model elements of modelling languages. These code structures are part of the program code in Figure 5.3.

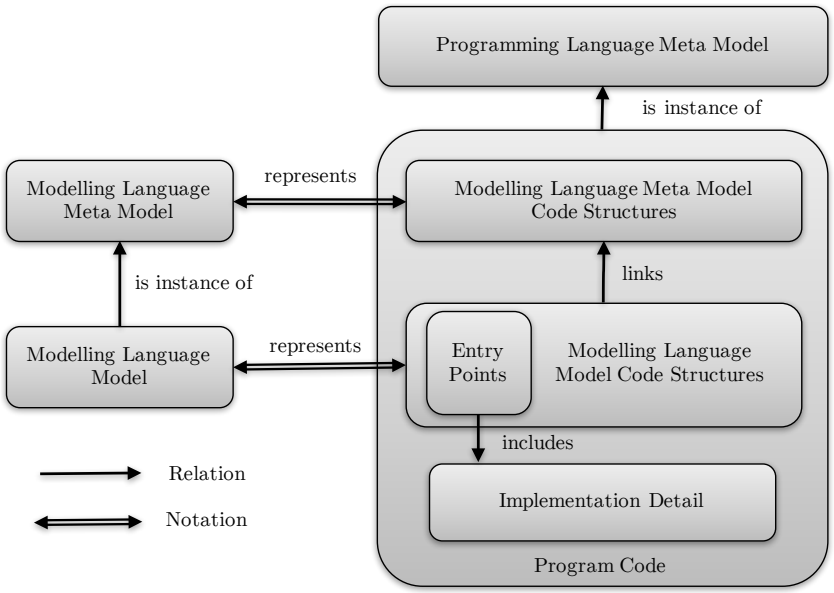


Figure 5.3: An overview of the elements in the Model Integration Concept and their interrelationships. Well-defined program code structures represent meta model elements and model elements of modelling languages. Entry points within the code structures can be used to enter arbitrary other code.

- *Modelling Language Model Code Structures:* Modelling language model code structures are parts of a program, that are defined to represent model elements of modelling languages. These code structures may link other code structures, which represent meta model elements of modelling languages, to define that a code structure represent an instance of the corresponding meta model element. These code structures are part of the program code in Figure 5.3.

- *Implementation Details:* Every program code, that does not represent a model element or a meta model element, is considered an implementation detail. Implementation details are not subject to translation in the approach presented in this thesis. Implementation details are part of the program code in Figure 5.3.

- *Entry Points:* Some parts of the code structures are fixed, so that the model-code mapping is well-defined. Other parts of the structures are explicitly declared to be flexible, so that a relationship to the rest of a program can be created. Entry points are these flexible places within modelling language model code structures, that can include implementation details. Entry points are part of the modelling language model code structures in Figure 5.3.

- *Notations:* Notations describe how a modelling language meta model element (*meta model notations*) or model element (*model notations*) are represented using program code structures written in a specific programming language. They describe an equivalence relation between meta model or model elements and program code structures. Notation are shown in Figure 5.3 as bidirectional edges with the label *represents* between the

modelling language meta model and the modelling language meta model code structures, and between the modelling language model and the modelling language model code structures.

- *Integration Mechanisms:* Integration mechanisms are a generalisation of notations. Integration mechanisms define a translation between placeholders of meta model elements or model elements and program code elements and can therefore be used as templates for building specific notations between languages. Integration mechanisms are not shown in Figure 5.3.

The following sections describe these elements and their interrelationships in more details, before they are defined in Section 5.4.

5.3.1 Modelling Language Meta Models

The abstract syntax of modelling languages in the context of the Model Integration Concept is described with meta models. These meta models define classes, attributes, and references. Classes own attributes and references. Attributes are typed. References define the allowed relationships between instances of classes. References target classes and have a cardinality. A subset of references are containment references, meaning that the reference's source owns the reference's target. Classes, attributes, and references are named. Figure 5.4 shows an example of a graphical notation[2] of a simple meta model with classes, attributes, and references. The nodes are classes with their name at the top and a list of attributes below. Attributes are notated with their name and their type. References are represented as edges between the nodes. Their name and cardinality is attached as label. Containment references have a black diamond at their source class. All references have an arrow head at their target class.

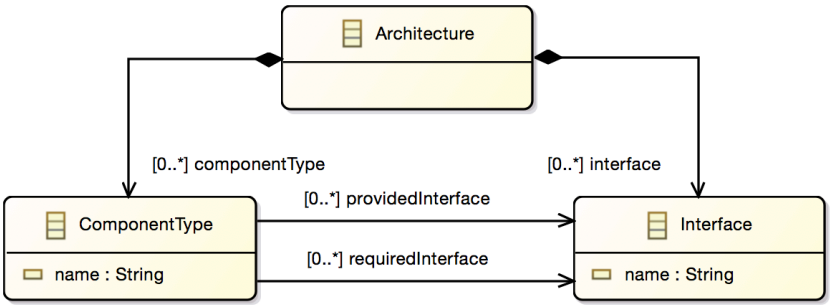


Figure 5.4: A graphical notation of a simple meta model

5.3.2 Modelling Language Models

Models are instances of meta models. They comprise objects, assign values to attributes and assign targets to references of objects. Figure 5.5 shows an example of a graphical notation of a

[2]The graphical notation for meta models used in this thesis is closely related to the one used by the EcoreTools (`https://www.eclipse.org/ecoretools/overview.html`) for Ecore meta models

simple model[3]. The model is an instance of the meta model in Figure 5.4. The nodes are object with the name of their class after a colon at the top, and a list of value assignments to attributes below. Attribute assignments show the name of the attribute of the class, an equality sign and the assigned value. The targets of references are represented as edges between the objects. Containment references have a black diamond at the source. Other references have an arrow head at the target. When a model is the instance of a meta model, it must be conform to the constraints regarding the classes, attributes, and references available, their typing, cardinalities, and naming, expressed by the meta model.

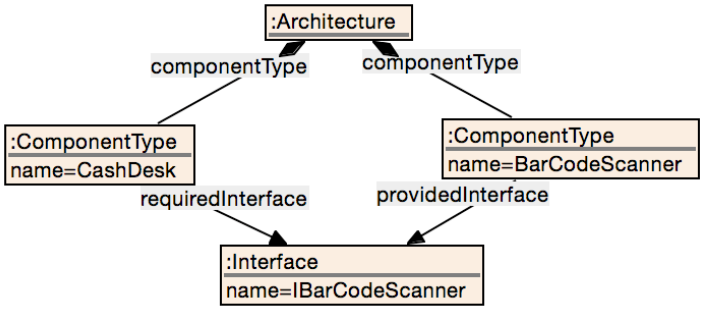


Figure 5.5: A graphical notation of a simple model. The model is an instance of the meta model shown in Figure 5.4.

5.3.3 Programming Language Meta Models

Programming languages in the context of the Model Integration Concept are defined like modelling languages, but with a specific set of abstract syntax elements and their interrelations in the meta model. Figure 5.6 shows an excerpt of the required abstract syntax. The excerpt defines the class *Type* which corresponds to an object-oriented type declaration. A type is named. An *Annotation* corresponds to annotations in Java. They are typed meta data declarations, that can be attached to other program code elements. Annotations are named, and can have *Annotation Parameters*. Annotation parameters are named and typed properties of annotations. When annotations are attached to an element, this can be represented using the class *AttachedAnnotation*. An attached annotation has an annotation as type and a list of *Attached Annotation Parameters* which correspond to the parameters of the attached annotation. The attached annotations parameters may have values corresponding to the types of their respective parameters[4].

The Model Integration Concept does not provide a thorough definition of a specific programming language, but describes abstract concepts, which need to be available in a programming language to be applicable in this context. A programming language is usable with the Model Integration Concept, when the necessary abstract syntax elements are mappable to abstract

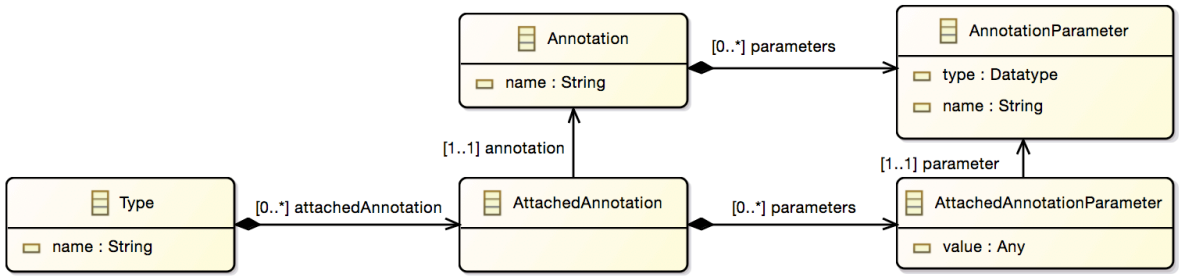


Figure 5.6: An excerpt of elements in the meta model of programming languages in the Model Integration Concept

syntax elements of the programming language.

5.3.4 Program Code

Programs are descriptions of structures and behaviour. They are typically notated with a textual syntax of a programming language. In the context of the Model Integration Concept, programs are models that instantiate programming language meta models. Figure 5.7 shows a very simple instance of the programming language meta model described above. Listing 5.1 shows this program in the Java programming language. It first declares the annotation `ComponentType` with a parameter `version`, a `String`. It then declares a type with the name `BarcodeScanner`, with the annotation `ComponentType` attached to it. The attachment assigns the value `1.0` to the parameter. In the context of this thesis, only descriptions of structures are considered. Imperative behaviour descriptions in program code, such as the content of operation bodies, are not within the scope of this thesis.

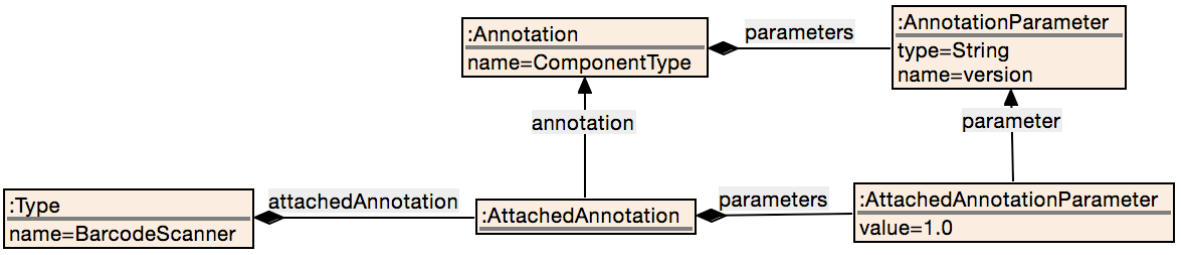


Figure 5.7: An example of a simple program

[3]The graphical notation for models in this thesis is the one used by the HenshinTGG Editor (`http://de-tu-berlin-tfs.github.io/Henshin-Editor/`)

[4]In the definition of programming languages in Section 5.4.4, attached annotation parameters can actually take multiple values for handling arrays. In this example, only single values are allowed.

```
public @interface ComponentType {
  String version();
}

@ComponentType(version="1.0")
public class BacodeScanner { }
```

Listing 5.1: The simple program of Figure 5.7 written in Java

5.3.5 Modelling Language Meta Model Code Structures

Modelling language meta model code structures are parts of program code, that represent meta model elements in the Model Integration Concept. Figure 5.8 shows an excerpt of the example of Figure 5.2. On the left side, it shows the class *ComponentType*, with its two attributes *name* and *version*, both of the type *String*. On the right side, the figure shows the code structure, that represents this meta model element in Java: an annotation declaration `ComponentType` with an annotation parameter `version` of the type `String`. The name attribute has no direct representation in the code structure.

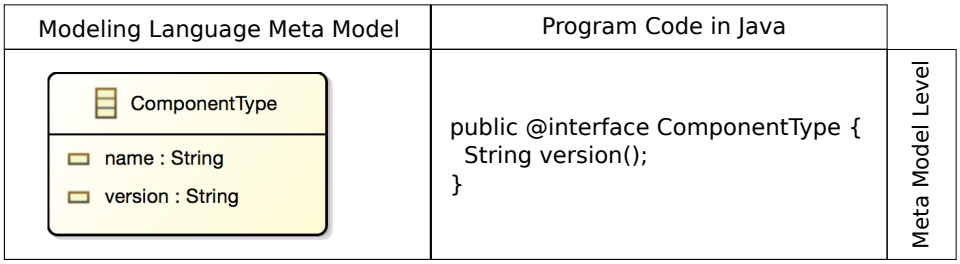


Figure 5.8: The meta model code structure (right hand side) extracted from the example of Figure 5.2

5.3.6 Modelling Language Model Code Structures

Modelling language model code structures are parts of program code, that represent model elements in the Model Integration Concept. Figure 5.9 shows an excerpt of the example of Figure 5.2. On the left side, it shows an object of the class *ComponentType*, which is shown in Figure 5.8. Its name attribute value is set to *BarcodeScanner*. Its version attribute value is set to *1.0*. On the right side, the figure shows the code structure, that represents this model element in Java: The type declaration `BarcodeScanner` represents the object. The annotation `ComponentType` is attached to the type, thus indicating, that the declared type represents an object of the class *ComponentType*. The name attribute's value corresponds to the declared type name. The version attribute's value is declared in the attached annotation parameter.

5.3.7 Notations

Notations define an equivalence relation between model elements and program code structures. There are two types of notations: *Meta model notations* define equivalence relationships be-

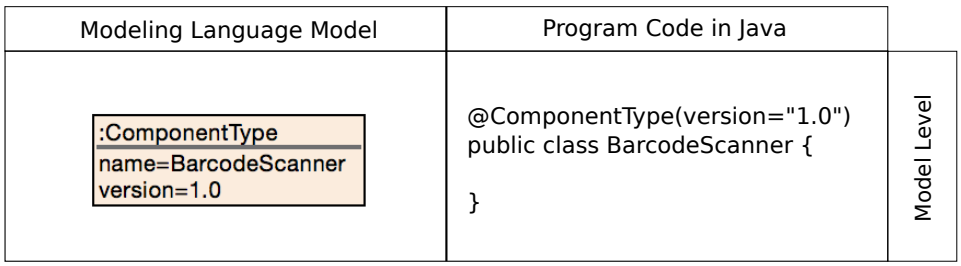


Figure 5.9: The model code structure (right hand side) extracted from the example of Figure 5.2

tween meta model elements and program code structures. *Model notations* define equivalence relationships between model elements and program code structures.

Figure 5.10 shows the meta model notation for the example meta model element and program code structure given in Figure 5.8. The attribute *name* is not declared in the meta model notation. This attribute is subject to the model notation.

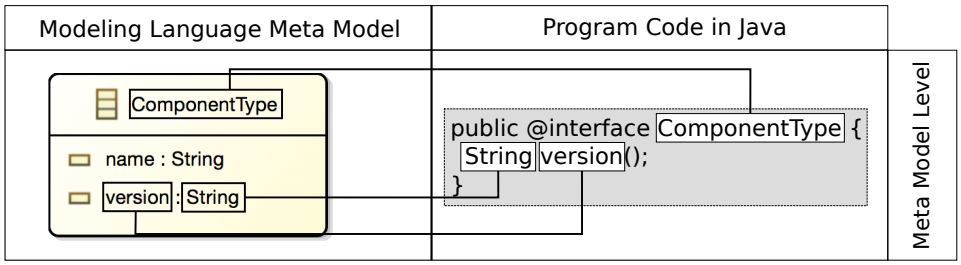


Figure 5.10: The meta model notation highlighted in the example of Figure 5.2

Figure 5.11 shows the model notation for the example model element and program code structure given in Figure 5.9.

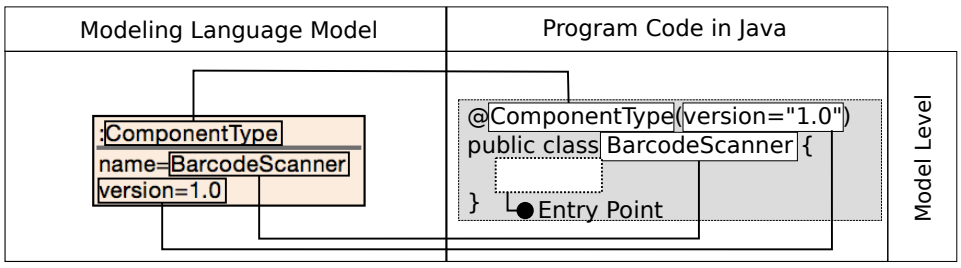


Figure 5.11: The model notation highlighted in the example of Figure 5.2

5.3.8 Entry Points

Entry points are parts of modelling language model code structures, that can include arbitrary other program code. Notations declare entry points in their program code structures. In Figure 5.11 the type declaration is considered the entry point in the code structure, indicated by the dashed box in the type's body. The entry point means that the type can be extended with arbitrary other code, e.g. further annotations, interface implementations, member attributes, or operations.

5.3.9 Integration Mechanisms

Integration mechanisms are templates for meta model notations and model notations. Instead of describing notations for specific meta model or model elements such as those described informally in the Figures 5.10 and 5.11, integration mechanisms describe notations for placeholder elements of meta models or models respectively. Several integration mechanisms are described in Section 5.6 for representing classes, attributes, containment references, or non-containment references.

This section gave an overview of the elements in the Model Integration Concept. In the next section, the elements will be described in detail.

5.4 Foundational Definitions

The Model Integration Concept defines formal bidirectional mappings between model and meta model elements on the one side and program code structures on the other side, using notations. Templates of actual notations will be presented later in this chapter. For understanding the formal definition of notations (see Section 5.5) and specific integration mechanisms (see Section 5.6), the foundations need to be formally defined. This section defines these foundational elements. First, the meta models (Section 5.4.1) and models (Section 5.4.2) of modelling languages are defined. Then, the meta model of programming languages (Section 5.4.4) and its instantiation (Section 5.4.5) are defined.

5.4.1 Modelling Language Meta Models

Meta models in the context of the Model Integration Concept describe the abstract syntax elements that can be modelled and the possible relations between these elements.[5] Figure 5.12 accompanies the definition as an overview of types of abstract syntax elements and their relationships to each other, to data types, and labels. The definitions 1 and 2 define meta models and their classes, attributes, and references.

[5] The concepts used for the modelling language meta model in this approach is conceptually based on the Ecore meta meta model [SBPM09].

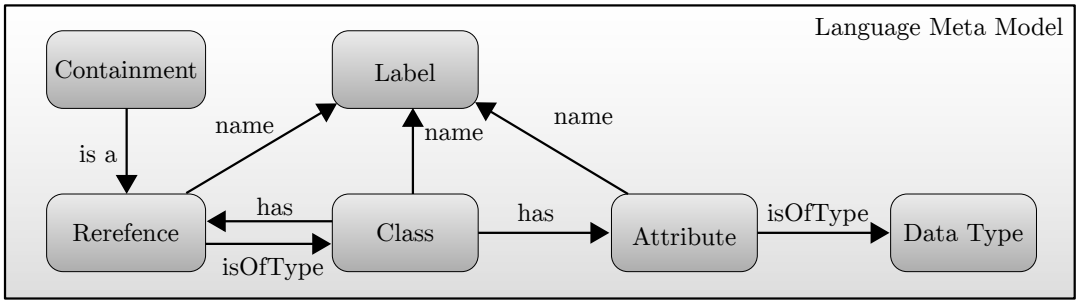


Figure 5.12: An overview of the elements of language meta models and their relations to each other

Definition 1: Modelling Language Meta Model

A meta model M_{Meta} is defined as follows:

$$M_{Meta} = (A, D, L, F), \text{ where}$$

- A is a set of abstract syntax elements of the language, with a typing function $\xrightarrow{isOf}: A \to \{ClassType, AttributeType, ReferenceType\}$,
- $D = \{String, Boolean, Int, Float, Void, Any, Datatype\}$ is a set of data types,
- L a set of labels,
- F is a set of relations and functions, that define the interdependencies between abstract syntax elements, data types, and string labels. The elements of F are subsequently defined in the following definitions.

Definition 2: Meta Model Abstract Syntax Element Types

For the abstract syntax elements A of a meta model, the following subsets of A are defined to highlight the types of abstract syntax elements:

- $Classes := \{a \mid a \in A \wedge a \xrightarrow{isOf} ClassType\}$,
- $Attributes := \{a \mid a \in A \wedge a \xrightarrow{isOf} AttributeType\}$,
- $References := \{a \mid a \in A \wedge a \xrightarrow{isOf} ReferenceType\}$.

The abstract syntax elements of language meta models are named:

Definition 3: Named Abstract Syntax Elements of Modelling Language Meta Models

For the abstract syntax elements A of a meta model and its labels L the function

$$name : A \to L$$

relates a name to an abstract syntax element.

Abstract syntax elements of meta models have interrelationships. Classes own attributes and references, which is declared with the relation $\xrightarrow{has}$. Attributes and References are typed using the function $\xrightarrow{isOfType}$. The type of attributes and references determine which kind of values they represent.

Definition 4: Classes own Attributes and References

The relation $\xrightarrow{has}$ defines that a class owns attributes and references:

$$\xrightarrow{has} \subseteq Classes \times (Attributes \cup References)$$

For a class $c \in Classes$ and an attribute or reference $e \in Attributes \cup References$, the relation $\xrightarrow{has}$ can also be notated $c \xrightarrow{has} e$. When an attribute or reference $e \in Attributes \cup References$ is owned by a class $c \in Classes$, the owned element can also be identified with a dot as separator:

$$c.e :\iff e, c \xrightarrow{has} e$$

Instead of the elements, their name can also be used. E.g. for a class $c \in Classes$ and an attribute $a \in Attributes$:

$$c \xrightarrow{has} a \wedge name(c) = \text{Component} \wedge name(a) = \text{id} \Leftrightarrow \text{Component.id} = a$$

Constraint 1: Constraints to Owning Attributes or References

Each attribute and each reference must be owned by exactly one class: For two classes $c_1, c_2 \in Classes$ and an attribute or reference $e \in Attributes \cup References$ the following must be true:

$$c_1 \xrightarrow{has} e \wedge c_2 \xrightarrow{has} e \implies c_1 = c_2$$

Attributes and references owned by the same class must not have the same name: For a class $c \in Classes$ and two attributes or references $e_1, e_2 \in Attributes \cup References$ the following must be true:

$$c \xrightarrow{has} e_1 \wedge c \xrightarrow{has} e_2 \wedge name(e_1) = name(e_2) \implies e_1 = e_2$$

Attributes and references are typed. The type of a reference is its targeted class.

Definition 5: Typed Attributes

For a meta model M_{Meta}, its attributes $Attributes$, and its types D (all but $Void$), the function

$$\xrightarrow{isOfType}: Attributes \rightarrow D \setminus \{Void\}$$

relates a data type to an attribute. $a \xrightarrow{isOfType} d$ can also be notated as $type(a) = d$.

Definition 6: Reference Targets

For a meta model M_{Meta}, its references $References$ and its classes $Classes$, the relation

$$\xrightarrow{isOfType} \subseteq References \times Classes$$

defines that a reference targets a class. $(r, c) \subseteq \xrightarrow{isOfType}$ can also be notated as $r \xrightarrow{isOfType} c$ or $type(r) = c$.

Please note that this allows a reference to have multiple types. This means that any object can be assigned as target, that instantiates one of these classes.

References have cardinalities, which define how many other objects an object can target with the reference.

Definition 7: Reference Cardinalities

For a meta model M_{Meta} and its references $References$, the cardinality of references is declared using the following function:

$$\xrightarrow{cardinality}: References \rightarrow \{0..1, 0..*, 1..1, 1..*\}$$

The cardinality constraints the amount of target objects in an instance of the meta model (see Constraint 6). If no cardinality is explicitly stated, a cardinality of $0..*$ is assumed.

A subset of references are containment references, meaning that the source object owns the target object.

Definition 8: Containment References

For a meta model M_{Meta} and its references $References$, the subset of references that are containment references is defined as:

$$Containment \subseteq References$$

Containment references define that instances of the origin class own the instances that are referenced by the containment reference.

5.4.2 Modelling Language Model

Meta models can be instantiated by models. When meta models are instantiated, instances of their classes are created, values are assigned to attributes, and targets are set for references, following the defined relations between abstract syntax elements of the meta model. Figure 5.13 accompanies the following definitions as an overview of the elements of models, their relationships to each other and to meta model elements. The upper part of Figure 5.13 shows the modeling language meta model elements defined above. The middle and the lower part show language models, their elements and their interconnections with each other and with a language meta model.

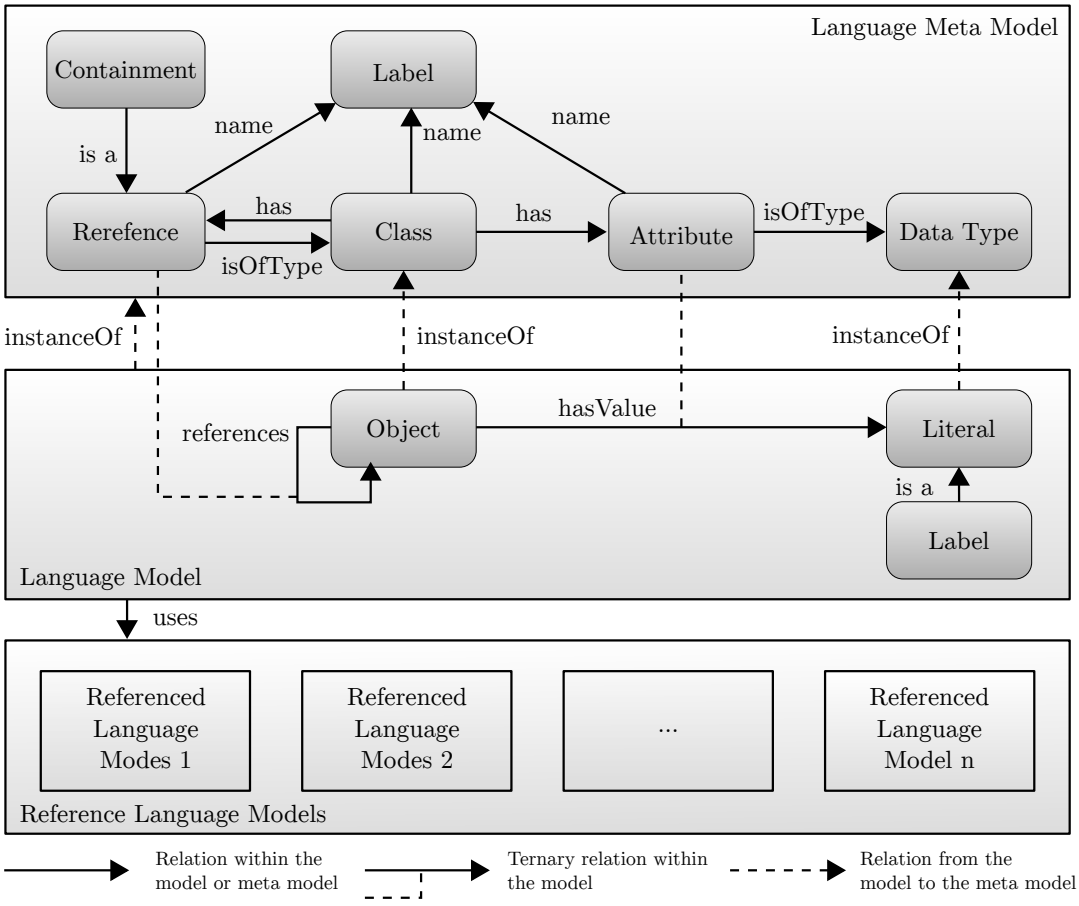


Figure 5.13: An overview of the elements of models, their relations to each other and to meta model elements

Definition 9: Modelling Language Model

A model M is an instance of a meta model M_{Meta}. It is defined as a tuple

$$M = (M_{Meta}, O, V, N, F, R), \text{ where}$$

- M_{Meta} is the meta model of the model,
- O is a set of objects building the model,
- V is a set of value literals, with a typing function $\xrightarrow{isOfType}: V \rightarrow D_{M_{Meta}}$
- $N \subseteq V$ is a set of labels,
- F is a set of relations and functions, that define the interconnection of the objects, data types of the meta model, value literals, and abstract syntax elements of the meta model, subsequently defined in the following definitions.
- R is a set of models which are referenced by this model.

A meta model is instantiated by defining objects, assigning attribute values, and targets to references of objects, in correspondence to the rules defined by the meta model.

Definition 10: Meta Model Instantiation

A model M instantiates a meta model M_{Meta} by instantiating the classes $Classes_{M_{Meta}}$ with objects O_M (see Definition 11), assigning values to attributes (see Definition 12), and assigning targets to references (see Definition 13).

Objects instantiate classes. They are typed by their classes.

Definition 11: Class Instantiation

An object instantiates a class. For a meta model, its classes $Classes$, a model of that meta model and its objects O, the function

$$\xrightarrow{instanceOf}: O \rightarrow Classes$$

returns the class that an object instantiates.

Values can be assigned to the attributes of an object's class.

Definition 12: Assigning Values to Attributes

For a meta model, its attributes $Attributes$, a model of that meta model, its objects O, and its value literals V, the relation

$$\xrightarrow{hasValue} \subseteq O \times Attributes \times V$$

assigns a value literal to an object's attribute. For an object $o \in O$, an attribute $a \in$

Attributes, and a value literal $v \in V$, this assignment can also be written:

$$o.a \xrightarrow{hasValue} v, \text{ or } value(o.a) = v$$

E.g. for an object $o \in O$, an attribute $a \in Attributes$ that is owned by the class that o instantiates (i.e. $o \xrightarrow{instanceOf} c \wedge c \xrightarrow{has} a$), with $name(a) = \text{id}$, and a value literal 1:

$$o.\text{id} \xrightarrow{hasValue} 1$$

Constraint 2: Values can only be Assigned to Attributes of an Object's Class

It is only possible to assign values to attributes for an object, when the attribute is owned by the object's class. Let o be an object, a be an attribute, and v be a value literal. Then

$$(o, a) \xrightarrow{hasValue} v \implies \exists c \in Classes : o \xrightarrow{instanceOf} c \wedge c \xrightarrow{has} a$$

Constraint 3: Value Assignments of Attributes must respect the Attribute's Type

The assignment of a value to an attribute for an object must respect the attribute's type. For a meta model, its class $c \in Classes$, its attribute $a \in Attributes$, its data type $d \in D$, a model of the meta model, and its object $o \in O$, the following must be true:

$$(o, a) \xrightarrow{hasValue} v \wedge o \xrightarrow{instanceOf} c \wedge c \xrightarrow{has} a \wedge a \xrightarrow{isOfType} d \implies v \xrightarrow{isOfType} d$$

Targets are assigned to references for objects. Target assignments respect the target type and the cardinality of the reference.

Definition 13: Assigning Targets to References

An object can assign targets to references owned by its class. For a meta model, its references $References$, a model of that meta model, and its objects O, the relation

$$\xrightarrow{references} \subseteq O \times References \times O$$

assigns a target to a reference for a source object.

Let M_{Meta} be a meta model, with $c_{source}, c_{target} \in Classes_{M_{Meta}}$ and $r \in References_{M_{Meta}}$. The reference is owned by c_{source} and targets c_{target}: $c_{source} \xrightarrow{has} r, r \xrightarrow{isOfType} c_{target}$.

Now let $M \xrightarrow{instanceOf} M_{Meta}$ be a model of that meta model, with two objects $o_{source}, o_{target} \in O_M$, with $o_{source} \xrightarrow{instanceOf} c_{source} \wedge o_{target} \xrightarrow{instanceOf} c_{target}$. Then the assignment $(o_{source}, r, o_{target}) \in \xrightarrow{references}$ can be written as follows:

$$(o_{source}, r) \xrightarrow{references} o_{target}$$

When the name of the reference is given, it can also be written in a short hand notation. Let the reference be named as follows: $name(r) = child$, then the assignment can be written as follows:

$$o_{source}.\text{child} \xrightarrow{references} o_{target}$$

Constraint 4: Targets can only be Assigned to References of an Object's Class

It is only possible to assign targets to references for an object, when the reference is owned by the object's class.

Let o_{source} be a source object, r be a reference, and o_{target} be a target object. Then

$$(o_{source}, r) \xrightarrow{references} o_{target} \implies \exists c \in Classes : o_{source} \xrightarrow{instanceOf} c \wedge c \xrightarrow{has} r$$

Constraint 5: Target Assignments of References must respect the Reference's Type

The assignment of targets to references must respect the reference's type. Let o_{source} be a source object, r be a reference, and o_{target} be a target object. Then

$$(o_{source}, r) \xrightarrow{references} o_{target} \implies \exists c \in Classes : o_{target} \xrightarrow{instanceOf} c \wedge r \xrightarrow{isOfType} c$$

Constraint 6: Target Assignments of References must respect the Reference's Cardinality

The assignment of targets to references must respect the reference's cardinality. Let s be a source object, and r be a reference. Let $\xrightarrow{references}_{s,r} \subseteq \xrightarrow{references}$ be the set of target assignments (s, r, o) for the reference r and the source object s. Then for all $\xrightarrow{references}_{s,r}$:

$$r \xrightarrow{cardinality} 0..1 \implies 0 \leq \left| \xrightarrow{references}_{s,r} \right| \leq 1$$

$$r \xrightarrow{cardinality} 0..* \implies 0 \leq \left| \xrightarrow{references}_{s,r} \right| \leq \infty$$

$$r \xrightarrow{cardinality} 1..* \implies 1 \leq \left| \xrightarrow{references}_{s,r} \right| \leq \infty$$

$$r \xrightarrow{cardinality} 1..1 \implies 1 \leq \left| \xrightarrow{references}_{s,r} \right| \leq 1$$

A model can reference other models. This means that all objects, literal values, string labels, and the functions and relations of the referenced models can be used in the referencing model.

Definition 14: Model Dependencies

For a model $M = (M_{Meta}, O, V, N, F, R)$, the set R declares models on which M depends. All objects $o \in O$, value literals $v \in V$, and functions and relations $f \in F$ of dependencies $r \in R$ can be used in the model M as if they were contained in the model itself.

Dependencies can span multiple levels. I.e. a model M_1 that depends on a model M_2 can itself be the dependency of another language M_0.

5.4.3 Example Meta Model and Example Model

The following examples show how this formalization can be used to describe a meta model and a corresponding model. Figure 5.14 depicts an example meta model. It defines a class *ComponentType* with two attributes: *name* and *version*, both of the type *String*. The component has a reference *provided* that targets the class *Interface*. The interface has an attribute named *name*, a *String*. The formalization of this meta model is given in Example 1.

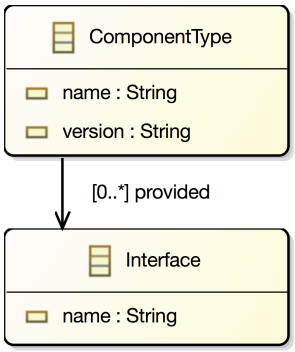


Figure 5.14: Example meta model as formalized in Example 1

Example 1: Formalization of the modelling language in Figure 5.14

The example meta model $M_{Meta}^{Example}$ is formalized as follows:

$$Classes := \{c, i\} \qquad Attributes := \{a_{version}, a^c_{name}, a^i_{name}\}$$
$$References := \{r_{provided}\} \qquad Containment := \emptyset$$

The elements are named as follows:

$$name(c) = \text{ ComponentType}, \qquad name(a_{version}) = \text{ version},$$
$$name(a^c_{name}) = \text{ name}, \qquad name(a^i_{name}) = \text{ name},$$

$$name(r_{provided}) = \text{ provided}$$

The attributes and references are defined as follows:

$$\text{ComponentType.version} \xrightarrow{isOfType} String, \qquad \text{ComponentType.name} \xrightarrow{isOfType} String,$$

$$\text{ComponentType.provided} \xrightarrow{references} \text{Component}, \qquad \text{InterfaceType.name} \xrightarrow{isOfType} String$$

Figure 5.15 shows a model of the meta model defined in Example 1. The model declares a component type with the name *BarcodeScanner* and the version *1.1*, which provides an interface *IBarcodeScanner*.

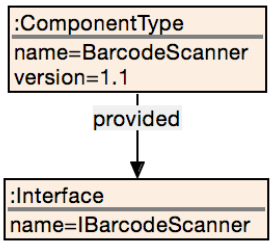


Figure 5.15: Example model as formalized in Example 2

Example 2: Formalization of a model based on the meta model defined in Example 1 as depicted in Figure 5.15

The example model $M^{Example} \xrightarrow{instanceOf} M^{Example}_{Meta}$ is formalized as follows:

$$O := \{c, i\}$$

The objects instantiate the following classes:

$$c \xrightarrow{instanceOf} ComponentType, \qquad i \xrightarrow{instanceOf} Interface$$

The assignments of values to attributes and targets of references are defined as follows:

$$i.\text{name} \xrightarrow{hasValue} \text{IBarcodeScanner}, \qquad c.\text{name} \xrightarrow{hasValue} \text{BarcodeScanner},$$

$$c.\text{version} \xrightarrow{hasValue} \text{1..1}, \qquad c.\text{provided} \xrightarrow{references} i$$

5.4.4 Programming Language Meta Models

Programming languages in the context of the Model Integration Concept contain specific abstract syntax elements and relationships between these elements, that are typical for current object oriented imperative programming languages. The definitions 15 to 28 specify the requirements for programming languages.

Definition 15: Programming Languages Meta Model

A programming language meta model P_{Meta} is similar to a meta model of modelling languages (see Definition 1) with the following constraints (Definitions 15 to 28). The following classes or equivalents must exist in $Classes_{P_{Meta}}$:

- $\mathcal{N}$ is the class of namespaces,
- $\mathcal{I}$ is the class of interfaces,
- $\mathcal{T}$ is the class of types,
- $\mathcal{MA}$ is the class of member attributes,
- $\mathcal{MR}$ is the class of member references,
- $\mathcal{OS}$ is the class of operation signatures,
- $\mathcal{OP}$ is the class of operation parameters,
- $\mathcal{O}$ is the class of operations,
- $\mathcal{A}$ is the class of annotations,
- $\mathcal{AP}$ is the class of annotation parameters,
- $\mathcal{AA}$ is the class of annotation attachments,
- $\mathcal{AAP}$ is the class of parameters of annotation attachments.

The classes $Classes_{P_{Meta}}$, their attributes $Attributes_{P_{Meta}}$ and their relations $References_{P_{Meta}}$, as well as the labels $L_{P_{Meta}}$, and the relations and functions $F_{P_{Meta}}$ will now be subsequently defined, including short examples. Figure 5.16 accompanies the definition as an overview of the abstract syntax elements and their containment references.

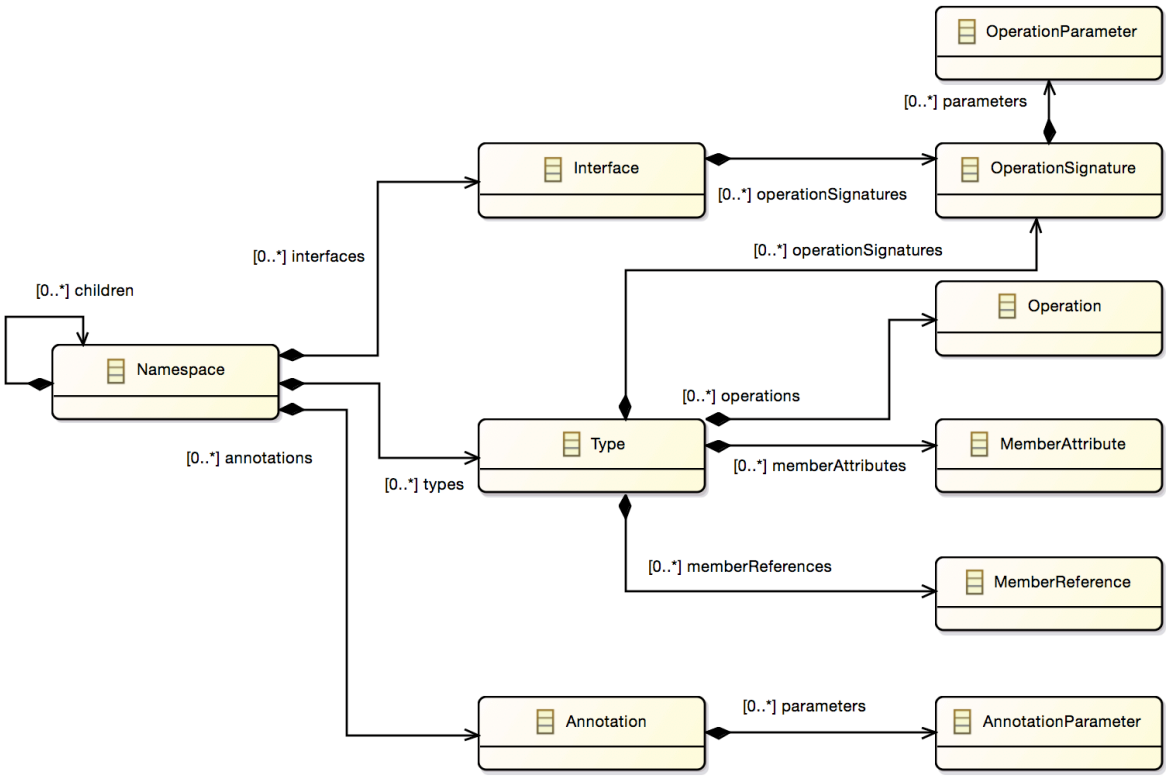


Figure 5.16: An overview of the types of abstract syntax elements in programming languages and their containment relations. Non-containment references and attributes are omitted here for readability reasons. Annotation attachments and their parameters are shown in Figure 5.22.

Named Elements

All classes of programming languages but operations, annotation attachments, and annotation attachment parameters own an attribute for assigning names to instances.

Definition 16: Named Elements in Programming Languages

Namespaces, interfaces, types, member attributes, member references, operation signatures, operation parameter, annotations, and annotation parameters each own an attribute *name* that is be used to assign names to their instances.

$$Named = \{\mathcal{N}, \mathcal{I}, \mathcal{T}, \mathcal{MA}, \mathcal{MR}, \mathcal{OS}, \mathcal{OP}, \mathcal{A}, \mathcal{AP}\}$$

$$\forall n \in Named : n.name \xrightarrow{isOfType} String$$

Namespaces

$\mathcal{N}$ is the class of namespaces. They own interfaces, types, and annotations. They are hierarchically organized, meaning that namespaces may own other namespaces.

Definition 17: Namespaces

For a programming language meta model, $\mathcal{N}$ is the class of namespaces. The following containment references declare the containment of interfaces, types, and annotations.

$$\mathcal{N}.interfaces \xrightarrow{isOfType} \mathcal{I},$$
$$\mathcal{N}.types \xrightarrow{isOfType} \mathcal{T},$$
$$\mathcal{N}.annotations \xrightarrow{isOfType} \mathcal{A}$$

$$interfaces, types, annotations \in Containment$$

Namespaces are hierarchically organized.

$$\mathcal{N}.children \xrightarrow{isOfType} \mathcal{N},$$

Interfaces and Types

$\mathcal{I}$ is the class of interfaces in programming languages. $\mathcal{T}$ is the class of types as they are known from object-oriented languages. Inheritance is not within the scope of this work. Types can implement interfaces. Figure 5.17 depicts the non-containment references of types and interfaces.

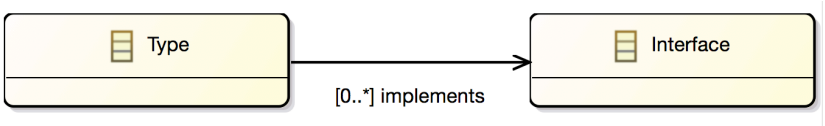


Figure 5.17: The non-containment references of types and interfaces. Types can implement interfaces.

Definition 18: Interfaces

$\mathcal{I}$ is the class of interfaces. Interfaces have operation signatures

$$\mathcal{I}.operationSignatures \xrightarrow{isOfType} \mathcal{OS},$$
$$operationSignatures \in Containment$$

Definition 19: Types

$\mathcal{T}$ is the class of types. Types have operation signatures, operations, member attributes, and member references.

$$\mathcal{T}.operationSignatures \xrightarrow{isOfType} \mathcal{OS},$$
$$\mathcal{T}.operations \xrightarrow{isOfType} \mathcal{O},$$
$$\mathcal{T}.memberAttributes \xrightarrow{isOfType} \mathcal{MA},$$
$$\mathcal{T}.memberReferences \xrightarrow{isOfType} \mathcal{MR},$$
$$operations \in Containment,$$
$$operationSignatures \in Containment,$$
$$memberAttributes \in Containment,$$
$$memberReferences \in Containment$$

Types can implement interfaces.

$$\mathcal{T}.implements \xrightarrow{isOfType} \mathcal{I}$$

Member Attributes and Member References

$\mathcal{MA}$ is the class of member attributes. Member attributes have types and may have values. Their type is a data type (*String*, *Boolean*, *Int* or *Float*). Figure 5.18 depicts the attributes of the model class *MemberAttribute*.

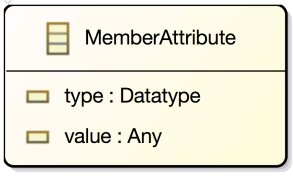


Figure 5.18: The attributes of the member attributes. Member attributes have types.

Definition 20: Member Attributes

Member attributes have types and optional values. The member attribute's type may be a data types of *String*, *Boolean*, *Int* or *Float* (notated with the type *Datatype*). The following attributes are defined for the class $\mathcal{MA}$:

$$\mathcal{MA}.type \xrightarrow{isOfType} Datatype,$$
$$\mathcal{MA}.value \xrightarrow{isOfType} Any$$

When an actual value is assigned to a member attribute, it must respect the type. The details on the constraints on assigning values to member attributes are defined in Constraint 12.

$\mathcal{MR}$ is the class of member references. Member references are typed by either an interface in $\mathcal{I}$ or a type in $\mathcal{T}$. They can be of an array type, meaning that they can reference multiple objects. Figure 5.19 depicts the attributes and non-containment references of the model class *MemberReference*.

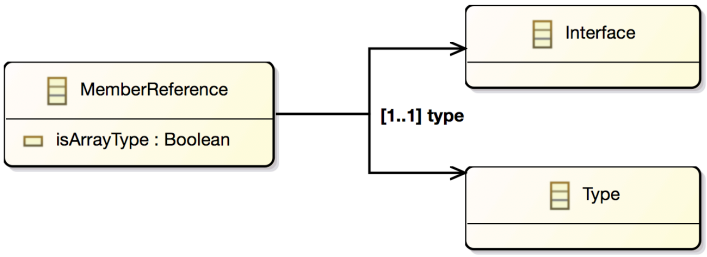


Figure 5.19: The attributes and non-containment references of member references. Member references are typed.

Definition 21: Member References

Member references have an interface or a type as target. They can be of an array type.

$$\mathcal{MR}.type \xrightarrow{isOfType} \mathcal{I},$$
$$\mathcal{MR}.type \xrightarrow{isOfType} \mathcal{T},$$
$$\mathcal{MR}.type \xrightarrow{cardinality} 1..1,$$
$$\mathcal{MR}.isArrayType \xrightarrow{isOfType} Boolean$$

If no value is explicitly assigned to the attribute $isArrayType$ for a member reference mr, it is assumed that $mr.isArrayType \xrightarrow{hasValue} false$.

Operation Signatures, Operation Parameters, and Operations

An operation is the implementation for an operation signature. Operation Signatures have operation parameters. Operation signatures and operation parameters are typed, either by a data type or by an interface. Figure 5.20 accompanies this definition as an overview.

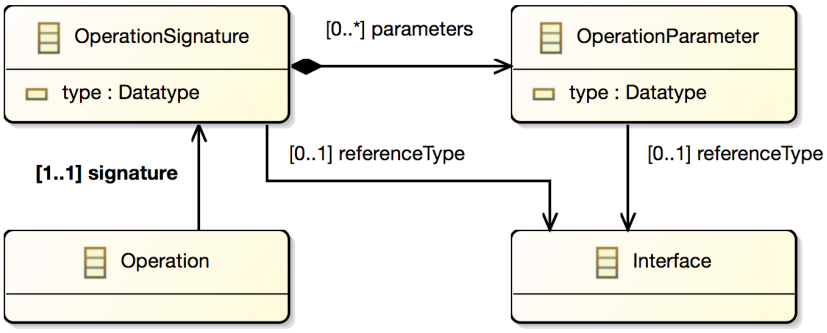


Figure 5.20: An overview of operations, operation signatures, and operation parameters

Definition 22: Operation Signatures

$\mathcal{OS}$ is the class of operations signatures. Interfaces and types have operation signatures (see definitions 18 and 19). Operation signatures have a set of operation parameters $\mathcal{OP}$.

$$\mathcal{OS}.parameters \xrightarrow{isOfType} \mathcal{OP},$$
$$parameters \in Containments$$

Operation Signatures have an object as type that instantiates an interface in $\mathcal{I}$ or a type in $\mathcal{T}$. It is considered the return type of the operation upon successful execution.

$$\mathcal{OS}.type \xrightarrow{isOfType} \mathcal{I},$$
$$\mathcal{OS}.type \xrightarrow{isOfType} \mathcal{T},$$
$$\mathcal{OS}.type \xrightarrow{cardinality} 1..1$$

Definition 23: Operation Parameters

$\mathcal{OP}$ is the class of operation parameters. Operation parameters have an object as type that instantiates an interface in $\mathcal{I}$ or a type in $\mathcal{T}$.

$$\mathcal{OP}.type \xrightarrow{isOfType} \mathcal{I},$$
$$\mathcal{OP}.type \xrightarrow{isOfType} \mathcal{T},$$
$$\mathcal{OP}.type \xrightarrow{cardinality} 1..1$$

Definition 24: Operations

$\mathcal{O}$ is the class of operations. An operation is the implementation for one operation signature.

$$\mathcal{O}.signature \xrightarrow{isOfType} \mathcal{OS},$$
$$\mathcal{O}.signature \xrightarrow{cardinality} 1..1$$

Constraint 7: Implementing Operation Signatures

Multiple operations can implement a signature. For each type, only one such operation must exist. Let $o_1, o_2 \in \mathcal{O}$ be operations, $os \in \mathcal{OS}$ an operation signature, and $t \in \mathcal{T}$ a type.

$$o_1.signature \xrightarrow{references} os \wedge o_2.signature \xrightarrow{references} os$$
$$\wedge t.operations \xrightarrow{references} o_1 \wedge t.operations \xrightarrow{references} o_2 \implies o_1 = o_2$$

Annotations

$\mathcal{A}$ is the class of typed meta data declarations called annotations. Annotations have typed parameters. Annotation parameters are typed analogously to operation signatures and operation parameters. Figure 5.21 accompanies this definition as an overview.

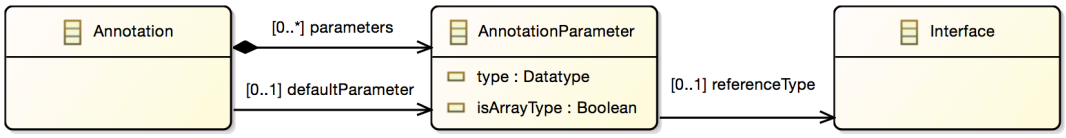


Figure 5.21: An overview of annotations and annotation parameters

Definition 25: Annotations

$\mathcal{A}$ is the class of annotations. They have annotation parameters:

$$\mathcal{A}.parameters \xrightarrow{isOfType} \mathcal{AP},$$
$$\mathcal{A}.parameters \in Containments$$

Definition 26: Annotation Parameters

$\mathcal{AP}$ is the class of annotation parameters. Annotation parameters may have a data type as type, or an object that instantiates an interface in $\mathcal{I}$ or a type in $\mathcal{T}$.

$$\mathcal{AP}.type \xrightarrow{isOfType} Datatype,$$
$$\mathcal{AP}.referenceType \xrightarrow{isOfType} \mathcal{I},$$
$$\mathcal{AP}.referenceType \xrightarrow{cardinality} 0..1$$

When the reference $referenceType$ is set, any value assigned to *type* is ignored. When neither the reference or attribute is set, any type can be assigned to the parameter as reference type (see Constraint 14).

Annotation parameters with a reference type can be of an array type. If no value is explicitly assigned to this attribute, it is assumed to be *false*.

$$\mathcal{AP}.isArrayType \xrightarrow{isOfType} Boolean$$

One annotation parameter of an annotation can be the default parameter.

$$\mathcal{A}.defaultParameter \xrightarrow{isOfType} \mathcal{AP}$$

$$\mathcal{A}.defaultParameter \xrightarrow{cardinality} 0..1$$

Constraint 8: Constraints to Array Type Annotation Parameters

Only annotation parameters with reference types can be of an array type.

$$\forall ap \in Classes_{P_{Meta}} : ap \xrightarrow{instanceOf} \mathcal{AP} \wedge ap.isArrayType \xrightarrow{hasValue} true \implies$$

$$\exists i \in Classes_{P_{Meta}} : i \xrightarrow{instanceOf} \mathcal{I} \wedge ap.referenceType \xrightarrow{hasValue} i$$

Annotations can be attached to types, interfaces, member attributes, member references, operations, and operation signatures. When these annotations are attached, values or targets have to be assigned to their parameters. Figure 5.22 gives an overview of the classes, references, and attributes for attaching annotations to elements and assigning values or targets to their parameters.

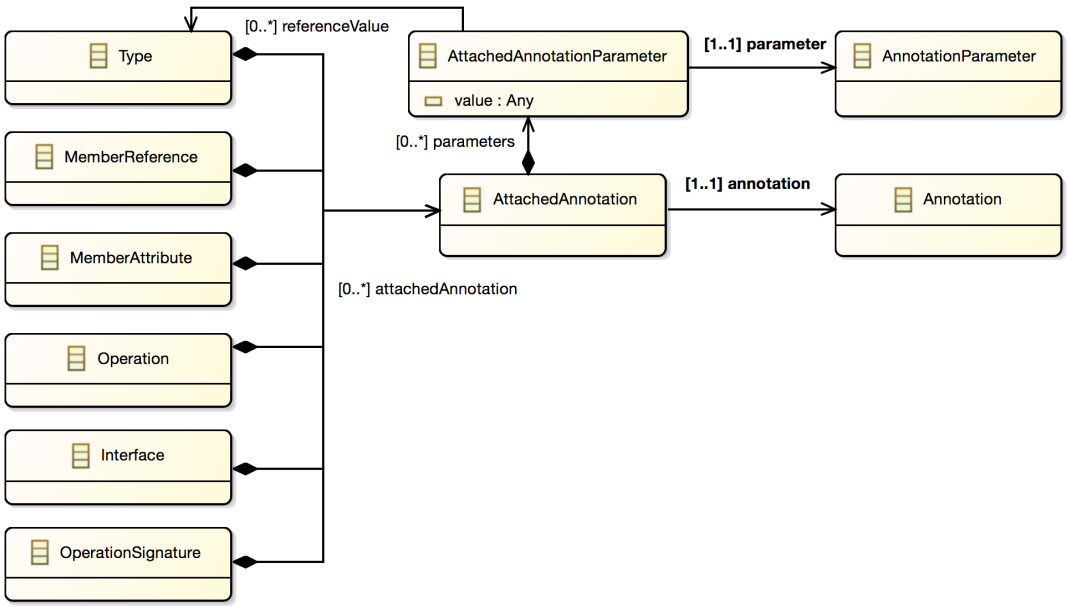


Figure 5.22: An overview of annotations and annotation parameters attached to elements

Definition 27: Annotation Attachments

$\mathcal{AA}$ is the class of annotation attachments. Annotation attachments are owned by types, interfaces, member attributes, member references, or operations.

$$\mathcal{T}.attachedAnnotations \xrightarrow{isOfType} \mathcal{AA}, \qquad \mathcal{T}.attachedAnnotations \in Containment,$$

$$\mathcal{I}.attachedAnnotations \xrightarrow{isOfType} \mathcal{AA}, \qquad \mathcal{I}.attachedAnnotations \in Containment,$$

$$\mathcal{MA}.attachedAnnotations \xrightarrow{isOfType} \mathcal{AA}, \quad \mathcal{MA}.attachedAnnotations \in Containment,$$

$$\mathcal{MR}.attachedAnnotations \xrightarrow{isOfType} \mathcal{AA}, \quad \mathcal{MR}.attachedAnnotations \in Containment,$$
$$\mathcal{O}.attachedAnnotations \xrightarrow{isOfType} \mathcal{AA}, \quad \mathcal{O}.attachedAnnotations \in Containment$$
$$\mathcal{OS}.attachedAnnotations \xrightarrow{isOfType} \mathcal{AA}, \quad \mathcal{OS}.attachedAnnotations \in Containment$$

Attached annotations reference an annotation that is attached to its owner.

$$\mathcal{AA}.annotation \xrightarrow{isOfType} \mathcal{A},$$
$$\mathcal{AA}.annotation \xrightarrow{cardinality} 1..1$$

Attached annotations may own attached annotation parameters.

$$\mathcal{AA}.parameters \xrightarrow{isOfType} \mathcal{AAP},$$
$$\mathcal{AA}.parameters \in Containment$$

Definition 28: Parameters of Annotation Attachments

$\mathcal{AAP}$ is the class of parameters for annotation attachments. They reference an annotation parameter. A value attribute and a value reference can be used to describe the static value of the attached annotation parameter.

$$\mathcal{AAP}.parameter \xrightarrow{isOfType} \mathcal{AP}, \quad \mathcal{AAP}.parameter \xrightarrow{cardinality} 1..1,$$
$$\mathcal{AAP}.value \xrightarrow{isOfType} Any, \quad \mathcal{AAP}.referenceValue \xrightarrow{isOfType} \mathcal{T}$$

When an actual value or reference value is assigned to a parameter of an attached annotation (see Definition 31), it must respect the parameter's type and cardinality. The details on the constraints on assigning values to parameters of attached annotations are defined in the Constraints 13 and 14.

5.4.5 Program Code

Program code is a model that instantiates a programming language meta model. In the following, program code and the constraints for creating program code are described.

Definition 29: Program Code Definition

Program code is an instance of a programming language meta model, following Definition 9 of models.

Constraint 9: Tree Structures for Namespaces

Structures of namespaces are only well-formed when they form a tree structure.

Member attributes and member references of types must be uniquely named.

Constraint 10: Uniquely Named Members within Types

The set M of member attributes $t.memberAttributes$ and member references $t.memberReferences$ of a type $t \in O, t \xrightarrow{instanceOf} \mathcal{T}$ must contain uniquely named objects.

$$\forall m^1, m^2 \in M : m^1.name = m^2.name \implies m^1 = m^2$$

Types that Implement Interfaces

Types can implement interfaces. When a type implements an interface, it has operations for each operation signature of the implemented interface.

Constraint 11: Interface Implementation

A type that implements an interface is obliged to own operations for all operation signatures of that interface.

For a program code, let $t \in O, t \xrightarrow{instanceOf} \mathcal{T}$ be a type and $i \in O, i \xrightarrow{instanceOf} \mathcal{I}$ be an interface. Then:

$$\begin{aligned} \forall os \in O : os \xrightarrow{instanceOf} \mathcal{OS} \wedge i.operationSignatures \xrightarrow{references} os \\ \wedge\, t.implements \xrightarrow{references} i \implies \exists o \in O \wedge o \xrightarrow{instanceOf} \mathcal{O} \\ \wedge\, t.operations \xrightarrow{references} o \wedge o.signature \xrightarrow{references} os \end{aligned}$$

Member Attribute Constraints

When assigning values to member attributes, the values must be of the member attribute's type.

Constraint 12: Member Attribute Value Assignments Respect their Type

A member attribute value must be an instance of the member attribute's type. I.e. for a data type $d \in D$, a member attribute $m \in O, m \xrightarrow{instanceOf} \mathcal{MA}$, and a value $v \in V$, the following must be true:

$$m.type \xrightarrow{hasValue} d \wedge m.value \xrightarrow{hasValue} v \implies v \xrightarrow{isOfType} d$$

Attaching Annotations to Elements

Annotations can be attached to types, interfaces, member attributes, member references, operation signatures, operation parameters, and operations.

Definition 30: Attaching Annotations to other Elements

The relation $\xrightarrow{attachedTo}$ is used for attaching annotations to elements. For an annotation $a \in O, a \xrightarrow{instanceOf} \mathcal{A}$ and an object $e \in O, e \xrightarrow{instanceOf} x, x \in \{\mathcal{T}, \mathcal{I}, \mathcal{MA}, \mathcal{MR}, \mathcal{O}, \mathcal{OS}\}$,

$$a \xrightarrow{attachedTo} e :\iff \exists \alpha \in O : \alpha \xrightarrow{instanceOf} \mathcal{AA} \\ \wedge\, e.attachedAnnotations \xrightarrow{references} \alpha \wedge \alpha.annotation \xrightarrow{references} a$$

When the name of the annotation is known, the attachment can also be written in a short hand notation. For an attachment $a \xrightarrow{attachedTo} e$, let the annotation be named as follows: $name(a) = \text{Versioned}$, then the assignment can be written as follows:

$$e.\text{Versioned}$$

Definition 31: Assigning Values to Parameters of Attached Annotations

When annotations are attached to elements, values must be assigned to the annotations' parameters for the attachment. The following relations is a short hand notation for assigning values to parameters in annotation attachments:

$$\xrightarrow{hasValue} \subseteq \Phi \times (V \cup Types)$$

The set Φ is the set of all tuples representing parameters of attached annotations:

$$\Phi := \{(e, a, ap) \mid a \xrightarrow{attachedTo} e \wedge a \xrightarrow{has} ap\}$$

The set $Types$ is the set of all types.

$$Types :\iff \{o \mid o \in O \wedge (o \xrightarrow{instanceOf} \mathcal{T})\}$$

The relation $\xrightarrow{hasValue}$ is defined as follows. Let e be an annotatable element (see Definition 30), $a \in O, a \xrightarrow{instanceOf} \mathcal{A}$ be an annotation, $ap \in O, ap \xrightarrow{instanceOf} \mathcal{AP}$ an annotation parameter, and $v \in V \cup Types$ a value literal or targeted type or interface object. Then let $a \xrightarrow{attachedTo} e$, and $\alpha \in O, \alpha \xrightarrow{instanceOf} \mathcal{AA}$ be the annotation attachment therein. Then:

$$(e, a, p) \xrightarrow{hasValue} v :\iff \exists \beta \in O \wedge \beta \xrightarrow{instanceOf} \mathcal{AAP} \\ \wedge\, \alpha.parameters \xrightarrow{references} \beta \wedge \beta.parameter \xrightarrow{references} p$$

The annotation parameter must be owned by the annotation.

$$(e, a, p) \xrightarrow{hasValue} v \implies a.parameters \xrightarrow{references} p$$

Let e be an annotatable element, a be an annotation, with $name(a) = \text{Versioned}$ and p a parameter of a, with $name(p) = \text{version}$, then the attached annotation parameter can also be identified as follows:

$$e.\text{Versioned.version}$$

Therefore the assignment of a value $v \in V \cup Types$ to the attached annotation parameter can be written as follows:

$$e.\text{Versioned.version} \xrightarrow{hasValue} v, \text{ or } value(e.\text{Versioned.version}) = v$$

Constraint 13: Annotation Parameter Value Assignments Respect their Cardinality

The allowed number of assigned targets for an annotation parameter is determined by the parameter's attribute $isArrayType$.

Let $\xrightarrow{hasValue}_{e,a,p} \subseteq \xrightarrow{hasValue}$ be the set of value assignments $(e, a, p) \xrightarrow{hasValue} v$. Then for all $\xrightarrow{hasValue}_{e,a,p}$:

$$p.isArrayType \xrightarrow{hasValue} false \implies 0 \leq \left| \xrightarrow{hasValue}_{e,a,p} \right| \leq 1$$

$$p.isArrayType \xrightarrow{hasValue} true \implies 0 \leq \left| \xrightarrow{hasValue}_{e,a,p} \right| \leq \infty$$

Constraint 14: Annotation Parameter Value Assignments Respect their Type

The value assigned to an annotation parameter must respect the parameter's type.

Let $(e, a, p) \xrightarrow{hasValue} v$ be an assignment of a value to an annotation parameter. When the annotation parameter is typed with a data type d, the value's type must match the respective data type.

$$p.\text{type} \xrightarrow{hasValue} d \wedge (e, a, p) \xrightarrow{hasValue} v \implies v \xrightarrow{instanceOf} d$$

When the annotation parameter is typed with an interface i, the assigned type t must implement the respective interface.

$$p.\text{type} \xrightarrow{hasValue} i \wedge (e, a, p) \xrightarrow{hasValue} v \implies v \xrightarrow{implements} i$$

When no type is declared for an annotation parameter, any type t can be assigned as value.

5.4.6 Example Program

The following examples show how this formalization can be used to represent a program. Listing 5.2 shows an annotation `ComponentType` in Java, that is included in a namespace with the name `componentmodel`. The annotation has an attribute `version` of the type `String`. Example 3 shows the formalization of that program.

```
package componentmodel;                                                    1
                                                                           2
public @interface ComponentType { String version(); }                      3
```

Listing 5.2: Example Annotation Declaration

Example 3: Formalization of the Java Code in Listing 5.2

The formalization for the Java code in Listing 5.2 is:

$$P_1 := (P_{Meta}, O, V, N, F, R), \text{ where } R = \emptyset,$$

$$O := \{n_{cm}, a_{ct}, ap_{version}\}$$

The objects instantiate the following classes:

$$c_{cm} \xrightarrow{instanceOf} \mathcal{N},$$
$$a_{ct} \xrightarrow{instanceOf} \mathcal{A},$$
$$a_{version} \xrightarrow{instanceOf} \mathcal{AP},$$

The elements are named as follows:

$$name(n_{cm}) = \text{componentmodel},$$
$$name(a_{ct}) = \text{ComponentType},$$
$$name(a_{version}) = \text{version}$$

The assignments of values to attributes and targets of references are defined as follows:

$$\text{componentmodel}.annotations \xrightarrow{references} \text{ComponentType},$$
$$\text{ComponentType}.parameters \xrightarrow{references} \text{version}$$

Listing 5.3 shows an attachment of the annotation that is declared in Listing 5.2. The code declares a type `BarcodeScanner` with a member attribute and two operations. The annotation `ComponentType` is attached to this type, with a value of "1.0" for the attribute parameter `version`. Example 4 shows the formalization of that program. Note that the statements within the operations are not formalized. Statements are not part of the formalization in the Model Integration Concept.

```
package cocome.barcodescanner;

@ComponentType(version="1.0")
public class BarcodeScanner {

  int scanned = 0;

  public void setScanned(int scanned){
    this.scanned = scanned;
  }

  public int getScanned(){
    return scanned;
  }
}
```

Listing 5.3: Example Type Declaration

Example 4: Formalization of the Java Code in Listing 5.3

The formalization for the Java code in Listing 5.3 is:

$$P_2 := (P_{Meta}, O, V, N, F, R), \text{ where } R = \{P_1\},$$

$$O := \{n_{cocome}, n_{bcs}, t_{bcs}, ma_{scanned}, os_{setScanned}, o_{setScanned}, op_{scanned}, os_{getScanned}, o_{getScanned}, aa, aap\}$$

The objects instantiate the following classes:

$$n_{cocome} \xrightarrow{instanceOf} \mathcal{N}, n_{bcs} \xrightarrow{instanceOf} \mathcal{N}, t_{bcs} \xrightarrow{instanceOf} \mathcal{T},$$
$$ma_{scanned} \xrightarrow{instanceOf} \mathcal{MA}, os_{setScanned} \xrightarrow{instanceOf} \mathcal{OS}, o_{setScanned} \xrightarrow{instanceOf} \mathcal{O},$$
$$op_{scanned} \xrightarrow{instanceOf} \mathcal{OP}, os_{getScanned} \xrightarrow{instanceOf} \mathcal{OS}, o_{getScanned} \xrightarrow{instanceOf} \mathcal{O},$$
$$aa \xrightarrow{instanceOf} \mathcal{AA}, aap \xrightarrow{instanceOf} \mathcal{AAP}$$

The elements are named as follows:

$$name(n_{cocome}) = \text{cocome}, \quad name(n_{bcs}) = \text{barcodescanner},$$
$$name(t_{bcs}) = \text{BarcodeScanner}, \quad name(ma_{scanned}) = \text{scanned},$$
$$name(os_{setScanned}) = \text{setScanned}, \quad name(op_{setScanned}) = \text{scanned},$$
$$name(os_{getScanned}) = \text{getScanned}$$

The assignments of values to attributes and targets of references are defined as follows:

$$\text{cocome}.children \xrightarrow{references} \text{barcodescanner},$$
$$\text{barcodescanner}.types \xrightarrow{references} \text{BarcodeScanner},$$
$$\text{BarcodeScanner}.operationSignatures \xrightarrow{references} \text{setScanned},$$

$$\text{BarcodeScanner}.operations \xrightarrow{references} o_{setScanned},$$
$$\text{BarcodeScanner}.memberAttributes \xrightarrow{references} \text{scanned},$$
$$\text{BarcodeScanner.ComponentType.version} \xrightarrow{hasValue} 1.0,$$
$$ma_{scanned}.type \xrightarrow{hasValue} Int,$$
$$ma_{scanned}.value \xrightarrow{hasValue} 0,$$
$$\text{setScanned}.parameters \xrightarrow{references} op_{scanned},$$
$$\text{setScanned}.type \xrightarrow{hasValue} Void,$$
$$op_{scanned}.type \xrightarrow{hasValue} Int,$$
$$o_{setScanned}.operationSignature \xrightarrow{references} \text{setScanned},$$
$$\text{getScanned}.type \xrightarrow{hasValue} Int,$$
$$o_{getScanned}.operationSignature \xrightarrow{references} \text{getScanned}$$

5.5 Notations

Notations provide means to represent model information with program code. The definition of notations in this thesis is conceptually based on the work of Moritz Balz [Bal11, p. 28].

5.5.1 Definition

There are two types of notations for representing model information in program code. The first type of notations is used to represent meta model elements in program code structures. This type of notations is called *meta model notation*.

Definition 32: Meta Model Notations

A meta model notation is a bidirectional mapping between modelling language meta model elements and specific program code structures. For the set of all program codes $\mathbb{P}$ and the set of all modelling language meta models $\mathbb{M}_{Meta}$, the relation of meta model notations is defined as:

$$\overset{represents}{\Longleftrightarrow} \subseteq \mathbb{P} \times \mathbb{M}_{Meta}$$

To emphasize the translational character, for a program P and a meta model M_{Meta}, the meta model notation can be declared as:

$$P \overset{represents}{\underset{\$name}{\Longleftrightarrow}} M_{Meta}$$

The notation's name *$name* is a variable that can be exchanged with a label to distinguish different specific notations.

The second type of notations is used to represent a model using program code structures. This type of notations is called *model notation*. Program code structures in model notations represent model elements. They use the code structures of the meta model notations to show

which meta model element is instantiated by the represented model element. Both notations work bidirectional. Using these notations, meta models and models can be translated into a program code representation and back.

Model notations also declare the set E of entry points. Entry points are elements that can be extended with arbitrary program code that is not part of the model notation. I.e. it is expected that these entry points contain code, e.g. for defining execution semantics of operations or fine-grained structures of types. This code can also be part of another model notation.

Definition 33: Model Notations

A model notation is a bidirectional mapping between model elements and program code structures. For the set of all programs $\mathbb{P}$, the powerset $2^{O_{\mathbb{P}}}$ of the set of all objects in $\mathbb{P}$, and the set of all modelling language models $\mathbb{M}$, the relation of model notations is defined as:

$$\overset{represents}{\longleftrightarrow} \subseteq \mathbb{P} \times 2^{O_{\mathbb{P}}} \times \mathbb{M}$$

For a given program P, the set E is a set of objects in P that serve as *entry points* in model notations. To emphasize the translational character, for a program P, a subset $E \subseteq O_P$ of objects in P and a model M, the model notation can be declared as:

$$(P, E) \underset{\$name}{\overset{represents}{\longleftrightarrow}} M$$

The notation's name *$name* is a variable that can be exchanged with a label to distinguish different specific notations.

5.5.2 Example

Figure 5.23 shows an example of notations. The figure shows a meta model and a model on the left side, and program code structures representing the meta model and model elements on the right side. The upper part of the figure describes a meta model notation $P \underset{ComponentType}{\overset{represents}{\longleftrightarrow}} Java$, for the program code $P \xrightarrow{instanceOf} Java$ and a modelling language M_{Meta}. *Java* is the programming language Java in the terms of the definition 15. The meta model defines a class *ComponentType*, which has the attributes *name* and *version*, both of the type *String*. The corresponding program defines an annotation with the name of the class, and an annotation parameter with the name and the type of the attribute *version*.

The lower part of the figure describes a model notation $(P, E) \underset{ComponentType}{\overset{represents}{\longleftrightarrow}} M$, for the program code $P \xrightarrow{instanceOf} Java$, an entry point E, and a model $M \xrightarrow{instanceOf} M_{Meta}$. The model consists of an object of the type *ComponentType*. Its name is *BarcodeScanner*. Its attribute *version* has the value "1.0". The corresponding program code defines a type with the name of the model's *ComponentType* object. The value of the *name* attribute is notated using the declared type name. The annotation defined in the meta model notation is attached to the type declaration. The value of the *version* attribute is given as the corresponding annotation parameter value. The type is declared to be the entry point of the notation. The type therefore allows for a structural and behavioural refinement of the model's *ComponentType* object. The

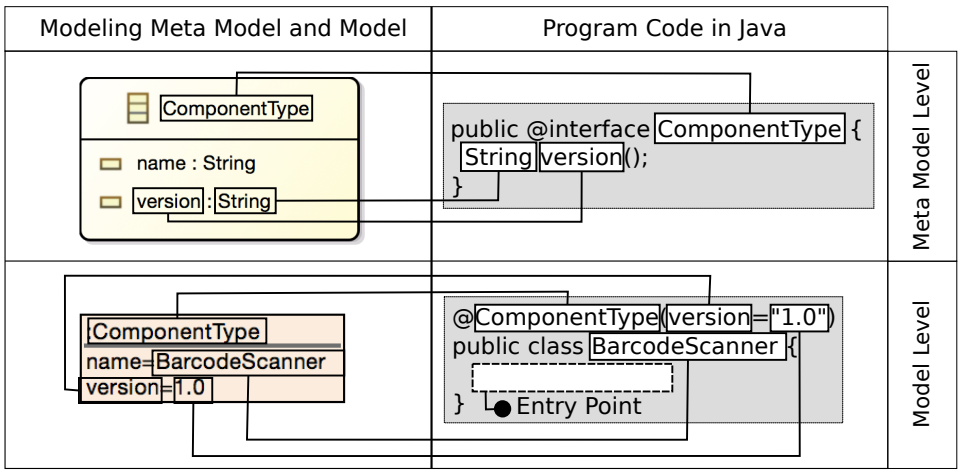


Figure 5.23: Example of a meta model and a model, notated with the programming language Java. The left side shows the meta model of the modelling language and the model. The right side shows possible program code structures for the meta model and the model in the programming language.

examples 5 and 6 describe the formalization of these notations. With such notations, meta models and models can be integrated with program code. The program code therefore can be translated into the corresponding meta model and model, and back.

Example 5: Example Meta Model Notation

The *ComponentType* meta model notation $P \xleftrightarrow[ComponentType]{represents} M_{Meta}$ is defined as follows[a]: The meta model M_{Meta} comprises a class and two attributes.

$$Classes_{M_{Meta}} = \{c\}, Attributes_{M_{Meta}} = \{a_n, a_v\}$$

The elements of the meta model are named as follows:

$$name(c) = \text{ComponentType}, name(a_n) = \text{name}, name(a_v) = \text{version}$$

The attributes and references of the meta model are defined as follows:

$$\text{ComponentType.name} \xrightarrow{isOfType} String,$$
$$\text{ComponentType.version} \xrightarrow{isOfType} String$$

The program comprises an annotation and an annotation parameter owned by that annotation.

$$O_P = \{a, ap\}, a \xrightarrow{instanceOf} \mathcal{A}, ap \xrightarrow{instanceOf} \mathcal{AP}, F_P = \{a \xrightarrow{has} ap\}$$

The annotation is named after the meta model's class name. The annotation parameter's name and type equal the name and type of the meta model's attribute *version*.

$$a.\text{name} = name(c), ap.\text{name} = name(a_v), ap.\text{type} = type(a_v)$$

[a] This and further definitions are separated into multiple blocks, with a description for each block for convenience.

Example 6: Example Model Notation

For a program P, and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Example 5, the *ComponentType* model notation $(P, E) \xleftrightarrow[ComponentType]{represents} M$ is defined as follows:

The model M comprises one object of the class named $ComponentType$ of the meta model M_{Meta}.

$$componentType \in O_M, componentType \xrightarrow{instanceOf} ComponentType$$

The attribute values and reference targets of the model are defined as follows:

$$componentType.\text{name} \xrightarrow{hasValue} \text{BarcodeScanner},$$
$$componentType.\text{version} \xrightarrow{hasValue} 1.0$$

Program structures P_{lib} exists that represent the meta model M_{Meta} with the *ComponentModel* meta model notation. The program P depends on these program structures.

$$P_{lib} \xleftrightarrow[ComponentType]{represents} M_{Meta}, P_{lib} \in R_P$$

The program code comprises a type.

$$t \in O_P, t \xrightarrow{instanceOf} \mathcal{T}$$

The type t is named after the value of the model's attribute $componentType$.name. The annotation $ComponentType$ is attached to the type t. The annotation parameter $version$ of the attached annotation has the same value as the model's attribute *version*.

$$t.\text{name} \xrightarrow{hasValue} value(componentType.\text{name}),$$
$$t.\text{ComponentType.version} \xrightarrow{hasValue} value(componentType.\text{version})$$

The model notation defines the body of the type t as entry point.

$$E = \{t\}$$

5.6 Integration Mechanisms

Integration mechanisms are templates for meta model notations and model notations. They describe a mapping between program code structures and symbolic meta model elements or symbolic model elements. Each comprises a meta model notation for translating a meta model element type and a corresponding model notation for translating instances of that element. Integration mechanisms can be instantiated by applying them to a specific meta model or model, i.e. by replacing the symbolic elements with specific elements.

Some integration mechanisms presented here have already been implicitly used in our previous publications [KKG14, Kon14, MBG11b, KG14, MBG11a]. Some are based on the work of Moritz Balz [Bal11]. Here, each integration mechanism is described using an example, formally defined, and then discussed. The descriptions are grouped by the type of meta model element that can be mapped with the respective integration mechanism.

The mechanisms are described based on a common running example, to emphasize how they can interact in a common meta model. First the running example meta model and model is shown in Section 5.6.1, before the mechanisms for classes (Section 5.6.2), containment references (Section 5.6.3), references in general (Section 5.6.4), and attributes (Section 5.6.5) are described.

5.6.1 Running Example

Each description of an integration mechanism will be accompanied by an excerpt of a running example. This section gives an overview of an example meta model and the example model (see Figure 5.24). The meta model comprises structural, behavioural, and quality-related aspects for software architectures. The following classes describe the structural aspects of the meta model.

Architecture The architecture is the root element of the model. It contains all structural elements and other elements that cover the overall architecture.

Interface An interface declares callable functionality. It can be provided or required by component types. An interface usually declares operations. These can be accessed by Transitions of State Machines via the interfaces. Interfaces are named.

ComponentType A component type represents an executable part of the software. It can provide or require interfaces. Component Types are named and versioned. They can be marked as being executable in parallel with themselves. Otherwise calls to their execution semantics will be sequentialized.

Namespace A namespace is used for organizing component types in a hierarchy. Components are only allowed to invoke operations on components that are in the same or a deeper scope. Namespaces are named.

The behavioural aspects of the meta model are described using the following classes:

StateMachine State machines are behavioural descriptions with a set of states and transition between them. They contain a set of variables.

Variables Variables are sets of arbitrary data that can be accessed by a state machine.

State A state machine is always in a defined state. These states are named. When the state is marked *immediate*, the state machine must not stay in this state, but immediately fire the next transition. In that case exactly one transition must be able to fire.

Transition A transition is a named behavioural step of a state machine between two states. During the transition a defined behaviour can be executed. Therefore the transition may have access to interfaces. Transitions have contracts they are obliged to fulfill.

Contract Before and after the execution of a transition, a contract is used to validate the preconditions and the post conditions for the transition, based on the machine's variables. Transitions can access interfaces and variables that may use data outside the definition of the state machine, e.g. because the data is provided by an external service. The compliance with the pre and post conditions can therefore usually not be statically checked in this model. Contracts are named.

Operation An operation describes executable semantics. Operations are named. Operations describe their expected time resource demand. It is possible to enter static values or probability functions regarding the time resource demand. Operations declare roles that are allowed to invoke them, and an execution strategy, how and where they are processed.

The further quality-related aspects of the meta model are described using the following classes:

Execution Strategy An execution strategy describes how and where an operation is to be processed. This could e.g. be the local processor for minor tasks, or a cloud environment for long running tasks.

Role A user of the system is always in one or more security roles. Roles are named.

The example model in the running example describes a simple architecture using these meta model elements. The example is based on the *CoCoME* benchmark program [HKW+08], but adjusted heavily for the means of this chapter. In the following, the foundational elements of the running example are described. Figure 5.25 shows the structural aspects of the model. The model's root object *Architecture* is not shown in the figure due to readability reasons. That element contains all component types, interfaces, state machines, roles, execution strategies in the model, and the root namespace. The structure consists of an architecture with four component types, that each provides an interface.

BarcodeScanner The bar code scanner scans bar codes and returns an identifier of the scanned product.

Printer The printer can print data to sheets of paper.

CashDesk The cash desk is used to handle sales. It uses the bar code scanner to scan items, and the printer to print a bill. The bar code scanner and the printer belong to the cash desk. They are therefore in a namespace hierarchy level below the cash desk.

StoreServer The store server gets information about sold items from the cash desk. It generates and provides access to a report of statistical data about the sold items. The store server can be executed in parallel with itself.

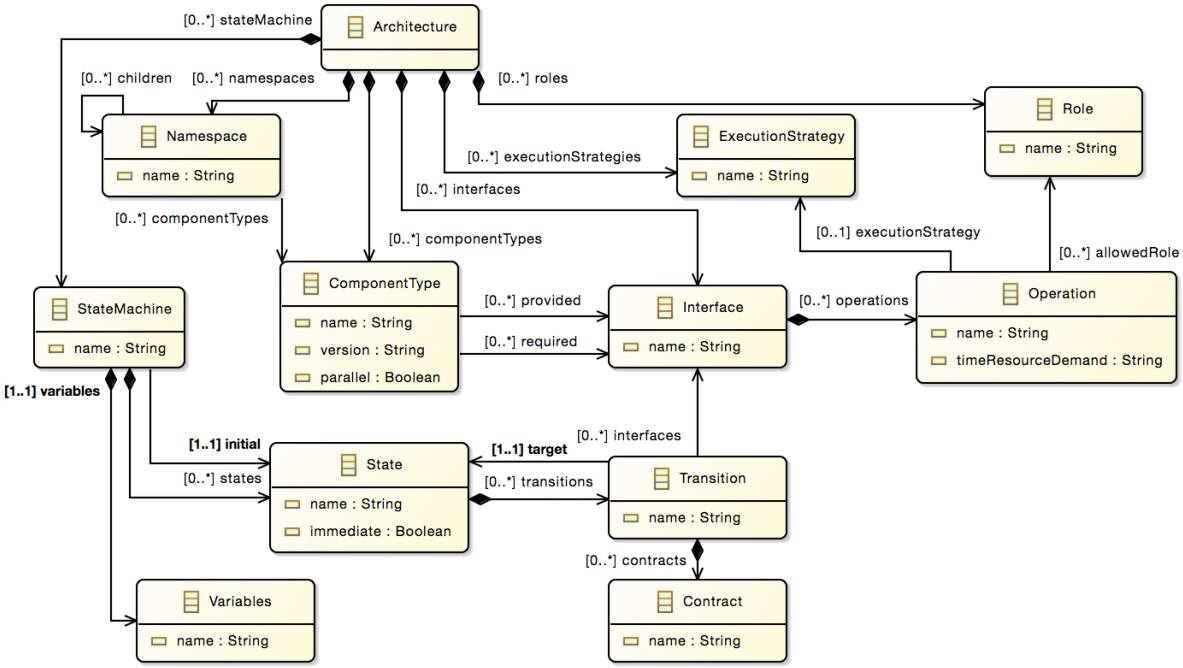


Figure 5.24: The meta model of the running example

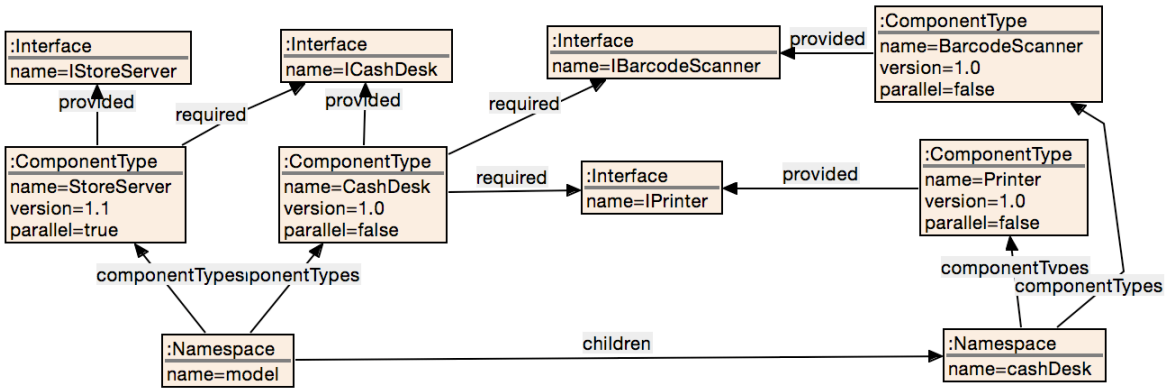


Figure 5.25: The structural aspects of the model in the running example

The behaviour of the component type *ICashDesk* is implemented with a state machine. Figure 5.26 shows the states and transitions of this state machine. The containment relationship of the state machine to the states is not shown in the figure due to readability reasons. The state machine comprises four states. The initial state is the state *Ready*. When an item is scanned, the state machine is in the state *Scanning*. From here further items can be scanned or the cash box can be opened. When the cash box is opened, the cash desk is *Awaiting Payment*. When the payment is received, the sale finishes, and the *CashDesk* is *Resetting*. This last state is an immediate state, meaning that the state machine will move forward with the next transition as soon as is comes into this state.

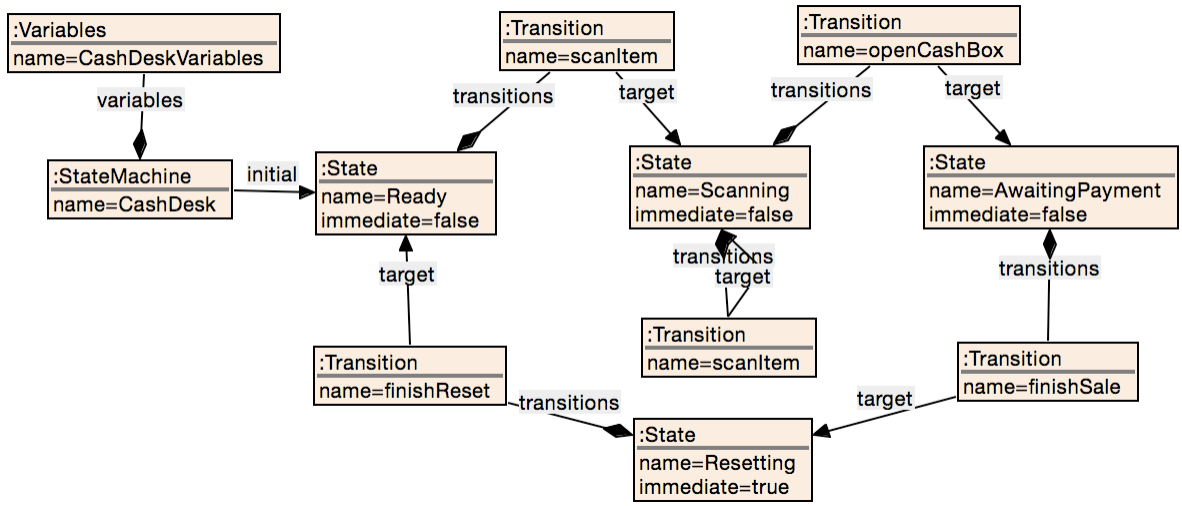


Figure 5.26: The state machine of the *CashDesk* component in the running example

Each class, attribute, and reference in the meta model of the running example is directly or indirectly mapped to an integration mechanism in the following sections. All identifying attributes are included in the mechanism mapped to their owning classes. Classes that are targets of the Containment Operation mechanism are mapped to this mechanism along with the corresponding containment reference. As an overview, Table 5.1 shows the mapping between meta model elements in the running example and the integration mechanisms shown in the following sections.

Meta Model Element	Integration Mechanism
Architecture	**Ninja Singleton**
→ all attributes and references	Included in the owner's mechanism
Namespace	**Namespace Hierarchy**
→ all attributes and references	Included in the owner's mechanism
ComponentType	**Type Annotation**
→ name	Included in the owner's mechanism
→ parallel	Constant Member Attribute
→ provided	Static Interface Implementation
→ required	Annotated Member Reference to Type Annotation or Static Interface
→ version	Attribute Annotation Parameter
Interface	**Static Interface**
→ name	Included in the owner's mechanism
→ operations	Containment Operation for Interfaces
Operation	**Contained in Interface.operations**
→ executionStrategy	Containment Operation Reference Annotation Parameter to Type Annotation or Static Interface for x..1 References
→ rolesAllowed	Containment Operation Reference Annotation Parameter to Type Annotation or Static Interface for x..* References
→ timeResourceDemand	Containment Operation Attribute Annotation Parameter
Role	**Type Annotation**
→ name	Included in the owner's mechanism
ExecutionStrategy	**Type Annotation**
→ name	Included in the owner's mechanism
StateMachine	**Type Annotation**
→ initial	Annotated Member Reference to Marker Interface for x..1 References
→ name	Included in the owner's mechanism
→ states	Annotated Member Reference to Marker Interface for x..* References
→ variables	Annotated Member Reference to Type Annotation or Static Interface
State	**Marker Interface**
→ immediate	Attribute Annotation
→ name	Included in the owner's mechanism
→ transitions	Containment Operation for Types
Transition	**Included in State.transition**
→ contracts	Containment Operation Reference Annotation Parameter to Marker Interface for x..* References
→ interfaces	Containment Operation Reference Annotation Parameter to Type Annotation or Static Interface for x..* References
→ name	Included in the owner's mechanism
→ target	Containment Operation Reference Annotation Parameter to Marker Interface for x..1 References
→ variables	Containment Operation Reference Annotation Parameter to Type Annotation or Static Interface for x..1 References
Contract	**Marker Interface**
→ name	Included in the owner's mechanism
Variables	**Static Interface**
→ name	Included in the owner's mechanism

Table 5.1: The running example's mapping of meta model elements and integration mechanisms

5.6.2 Class Representation

Integration mechanisms for class representations can be used to represent classes in a meta model and their objects in a model with program code structures. Table 5.2 lists integration mechanisms for representing classes and objects in models with program code structures.

Class Representations
Type Annotation
Marker Interface
Static Interface
Ninja Singleton
Namespace Hierarchy

Table 5.2: An overview of integration mechanisms for representing classes and objects with program code structures

Type Annotation

Using the Type Annotation mechanism, an annotation with the name of the class represents the class. The annotation is attached to a type, whose name equals the value of an identifying attribute of the represented class.

Example Figure 5.27 shows an example of the Type Annotation mechanism. The meta model specifies a class *ComponentType* with the attribute *name*, a *String*. The model instantiates this class with an object and assigns the value *BarcodeScanner* to the name attribute. The corresponding code defines the annotation `ComponentType`, and a type `BarcodeScanner` which has the annotation attached to it and therefore represents the component *BarcodeScanner*.

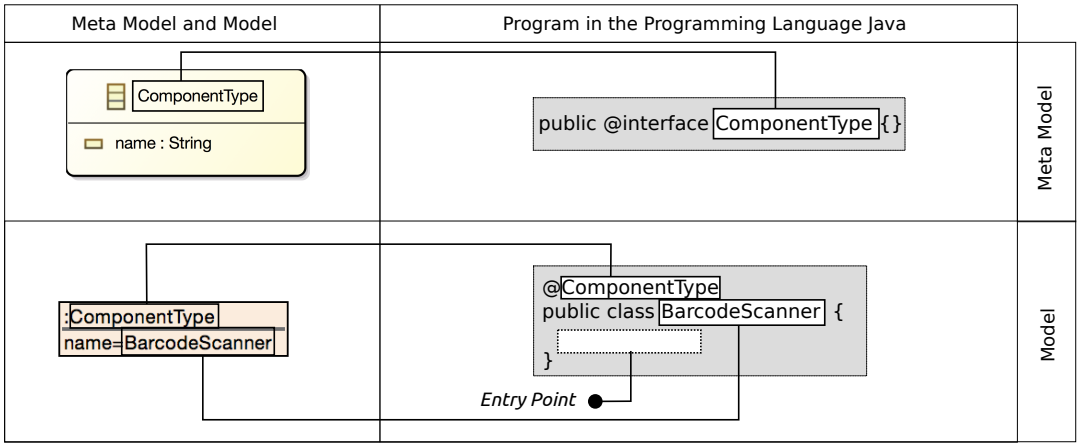


Figure 5.27: Example code for the Type Annotation Mechanism

Formalization The Type Annotation mechanism is formalized as follows:

Definition 34: Type Annotation - Meta Model Notation

The meta model notation of the Type Annotation mechanism $P \xleftrightarrow[TypeAnnotation]{represents} M_{Meta}$ is defined as follows:

The meta model M_{Meta} comprises a class and an attribute.

$$class \in Classes_{M_{Meta}}, id \in Attributes_{M_{Meta}}$$

The attribute of the meta model is defined as follows:

$$class.id \xrightarrow{isOfType} String$$

The program declares an annotation.

$$annotation \in O_P, annotation \xrightarrow{instanceOf} \mathcal{A}$$

The annotation is named after the meta model's class name.

$$annotation.name \xrightarrow{hasValue} name(class)$$

Definition 35: Type Annotation - Model Notation

For a program P and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Definition 34, the model notation of the Type Annotation mechanism $(P, E) \xleftarrow[TypeAnnotation]{represents} M$ is defined as follows:

The model M comprises one object of the class *class* of the meta model M_{Meta}.

$$object \in O_M, object \xrightarrow{instanceOf} class$$

Program code structures P_{lib} exists that represent the meta model M_{Meta} with the Type Annotation meta model notation. The program references these program structures.

$$P_{lib} \xleftrightarrow[TypeAnnotation]{represents} M_{Meta}, P_{lib} \in R_P$$

The program declares a type named after the value of the model's identifying attribute. The annotation of the meta model notation is attached to the type.

$$type \in O_P, type \xrightarrow{instanceOf} \mathcal{T} \qquad type.name \xrightarrow{hasValue} value(class.id),$$
$$annotation \xrightarrow{attachedTo} type$$

The model notation defines the body of the type as entry point.

$$E = \{type\}$$

Discussion For this mechanism to be applicable, the class to represent must have an identifying attribute of the type *String*, to be mappable to the name of the code element in programming languages. At design time, objects of the class can be identified by finding all types with the annotation attached. An execution runtime can instantiate such a type, execute its execution semantics, and make the instance available where necessary.

The entry point of the Type Annotation mechanism is a type body. It is therefore possible to include imperative execution semantics within standardized operations of that type. The Type Annotation mechanism is therefore good to represent classes that have individual behavioural execution semantics. There is no common denominator that enforces the existence of these operations. Therefore errors might occur, when an execution runtime tries to execute such an operation, which does not exist, e.g. because of an incorrectly spelled operation name.

The concept behind the type annotation mechanism is broadly known in the Java community from frameworks such as Enterprise JavaBeans (EJB) [EJB13] or the Contexts and Dependency Injection (CDI) [JSR14], they are also used in the Windows Workflow Foundations of the .Net Framework (e.g. [Mic]).

Marker Interface

Marker interfaces are interfaces that are used to mark a type to have certain properties. In the Marker Interface mechanism the interface represents the meta model's class. A type that implements the interface is marked as an object that instantiates the class. The name of the interface equals the name of the translated class, while the name of the marked type equals the value of an identifying attribute.

Example Figure 5.28 shows an example of the Marker Interface mechanism. The meta model specifies a class *State* with the attribute *name*, a *String*. The model instantiates this class with an object, and assigns the value *Ready* to the attribute *name*. The corresponding code declares the marker interface State, and a type Ready that implements the marker interface and therefore represents the state object.

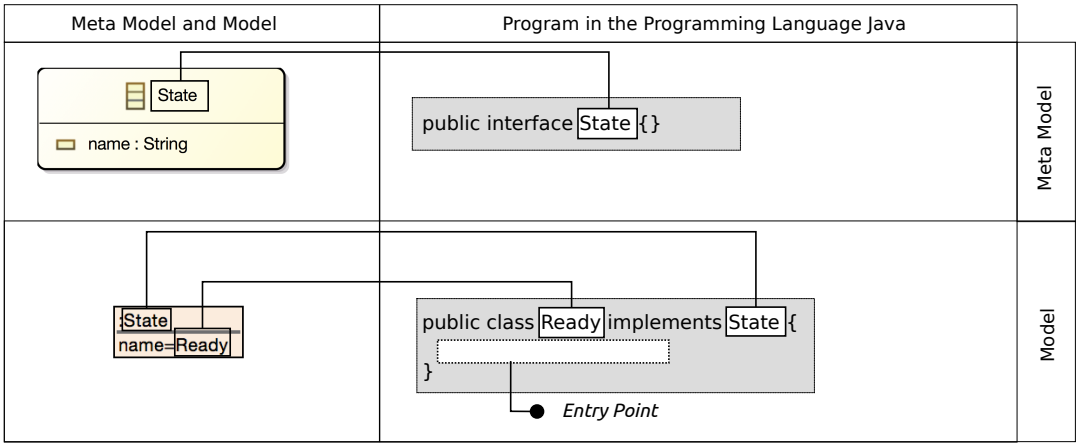


Figure 5.28: Example code for the Marker Interface Mechanism

Formalization The Marker Interface mechanism is formalized as follows:

Definition 36: Marker Interface - Meta Model Notation

The meta model notation of the Marker Interface mechanism $P \xleftrightarrow[MarkerInterface]{represents} M_{Meta}$ is defined as follows:

The meta model M_{Meta} comprises a class and an attribute.

$$class \in Classes_{M_{Meta}}, id \in Attributes_{M_{Meta}}$$

The attributes and references of the meta model are defined as follows:

$$class.id \xrightarrow{isOfType} String$$

The program declares an interface, which is named after the meta model's class name.

$$interface \in O_P, interface \xrightarrow{instanceOf} \mathcal{I}$$
$$interface.name \xrightarrow{hasValue} name(class)$$

Definition 37: Marker Interface - Model Notation

For a program P and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Definition 36, the model notation of the Marker Interface mechanism $(P, E) \xleftarrow[MarkerInterface]{represents} M$ is defined as follows:

The model M comprises one object of the class *class* of the meta model M_{Meta}.

$$object \in O_M, object \xrightarrow{instanceOf} class$$

Program structures P_{lib} exists that represent the meta model M_{Meta} with the Marker Interface meta model notation. The program references these program structures.

$$P_{lib} \underset{MarkerInterface}{\overset{represents}{\Longleftrightarrow}} M_{Meta}, P_{lib} \in R_P$$

The program declares a type.

$$type \in O_P, type \xrightarrow{instanceOf} \mathcal{T}$$

The type is named after the value of the model's identifying attribute. It implements the marker interface.

$$type.name \xrightarrow{hasValue} value(object.id),$$
$$type \xrightarrow{implements} interface$$

The model notation defines the body of the type *type* as entry point.

$$E = \{type\}$$

Discussion For this mechanism to be applicable, the class to represent must have an identifying attribute of the type *String*, to be mappable to the name of the code element in programming languages. At design time, objects of the class can be identified by finding types that instantiate the marker interface. An execution runtime can instantiate such a type, execute its execution semantics, and make the instance available where necessary.

The entry point of the Marker Interface mechanism is a type body. Like in the Type Annotation mechanism, it is possible to include imperative execution semantics within operations.

In contrast to the Type Annotation mechanism, this mechanism uses a common denominator for describing execution semantics. Operation signatures can be declared in the marker interface, which makes it mandatory to implement the corresponding operations in the types that implement the interface. The operations are available to an execution runtime, and to arbitrary program code. The Marker Interface mechanism is therefore good to represent classes that have behavioural execution semantics.

The marker interface can also be used as a basis for references to objects of the class in the model. Mechanisms for representing references benefit from this property (e.g. see the mechanism Annotated Member Reference in the Definitions 48 to 53. The concept behind the Marker Interface mechanism is a common solution for marking types in Java [Blo08, p. 197].

Static Interface

Using the Static Interface mechanism, a class is represented by an annotation. Objects of the class are represented by interface declarations to which the annotation is attached.

Example Figure 5.29 shows an example of the Static Interface mechanism. The meta model specifies a class *Interface* with the attribute *name*, a *String*. The model instantiates this class with an object, and assigns the value *IBarcodeScanner* to the name attribute. The corresponding code declares the annotation `Interface` which is attached to an interface `IBarcodeScanner`. The interface in the program code represents the component interface *IBarcodeScanner*.

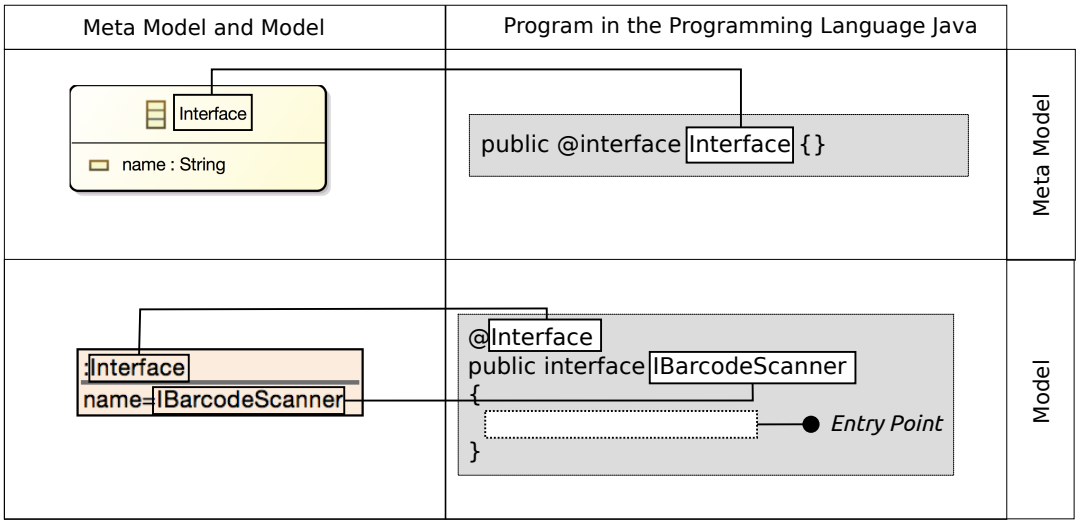


Figure 5.29: Example code for the Static Interface Mechanism

Formalization The Static Interface mechanism is formalized as follows:

Definition 38: Static Interface - Meta Model Notation

The meta model notation of the Static Interface mechanism $P \xleftrightarrow[StaticInterface]{represents} M_{Meta}$ is defined as follows:

The meta model M_{Meta} comprises a class and an attribute.

$$class \in Classes_{M_{Meta}}, a \in Attributes_{M_{Meta}}$$

The attributes and references of the meta model are defined as follows:

$$class.id \xrightarrow{isOfType} String$$

The program declares an annotation.

$$annotation \in O_P, annotation \xrightarrow{instanceOf} \mathcal{A}$$

The annotation is named after the meta model's class name.

$$annotation.\text{name} \xrightarrow{hasValue} name(class)$$

Definition 39: Static Interface - Model Notation

For a program P and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Definition 38, the model notation of the Static Interface mechanism $(P, E) \xleftarrow[StaticInterface]{represents} M$ is defined as follows:

The model M comprises one object of the class $class$ of the meta model M_{Meta}.

$$object \in O_M, object \xrightarrow{instanceOf} class$$

Program structures P_{lib} exists that represent the meta model M_{Meta} with the Static Interface meta model notation. The program P references these program structures.

$$P_{lib} \xleftrightarrow[StaticInterface]{represents} M_{Meta}, class \in Classes_{M_{Meta}}, class.id \in Attributes_{M_{Meta}},$$
$$annotation \in O_{P_{lib}}, annotation \xrightarrow{instanceOf} \mathcal{A}, P_{lib} \in R_P$$

The program declares an interface.

$$interface \in O_P, interface \xrightarrow{instanceOf} \mathcal{I}$$

The interface is named after the value of the identifying attribute of the represented class. The annotation is attached to the interface.

$$interface.name \xrightarrow{hasValue} value(object.id),$$
$$annotation \xrightarrow{attachedTo} interface$$

The model notation defines interface $interface$ as entry point, meaning that further annotation can be attached, and operation signatures can be added to the interface.

$$E = \{interface\}$$

Discussion For this mechanism to be applicable, the class to represent must have an identifying attribute of the type *String*, to be mappable to the name of the code element in programming languages. At design time, objects of the class can be identified by interfaces with their respective annotations. The entry point of the Static Interface mechanism is the interface body. It is therefore not possible to include imperative execution semantics within the code structures of the element, in contrast to the Marker Interface mechanism or the Marker Interface mechanism, where imperative execution semantics can be placed within operations.

It is, however, possible to add operation signatures that declare callable semantics. The Static Interface Implementation mechanism (see Definitions 54 and 55) declares a type that implements the interface of the Static Interface mechanism. This type is forced to implement

operations for all signatures declared in the interface, and therefore to provide execution semantics declared by the interface. In a variant of the Annotated Member Reference mechanism a member reference is created using the static interface as a type (see Definition 49). This allows to execute the declared execution semantics using operation calls.

The Static Interface mechanism is therefore good to represent classes that have no behavioural execution semantics for themselves, but declare them for others. This is a common pattern for the provision and requirement of component or service interfaces, where providers implement, and users reference interfaces. When semantics are declared with this mechanism, they can be provided with the Static Interface mechanism, and consumed with any other containment or non-containment reference mechanism. Examples for this pattern can be found e.g. in the OSGi framework for services [The14, Section 5.2].

Ninja Singleton

Using the Ninja Singleton mechanism, a class is not represented by specific code structures. Instead, the existence of exactly one instance of the class is assumed. This is useful for singleton objects in models, e.g. a root object *architecture* of an architectural description, that is the container of a variety of other elements.

Example Figure 5.30 shows an example of the Ninja Singleton mechanism. The meta model specifies a class *Architecture* without attributes or references. The model instantiates this class with an object. No corresponding code exists.

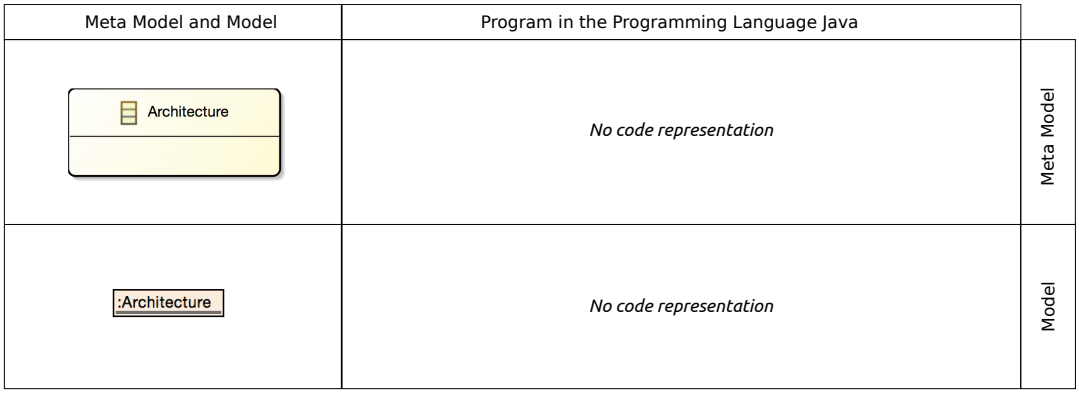


Figure 5.30: Example code for the Ninja Singleton Mechanism

Formalization The Ninja Singleton mechanism is formalized as follows:

Definition 40: Ninja Singleton - Meta Model Notation

The meta model notation of the Ninja Singleton mechanism $P \xleftrightarrow[NinjaSingleton]{represents} M_{Meta}$ is defined as follows:

The meta model M_{Meta} comprises a class.

$$class \in Classes_{M_{Meta}}$$

Definition 41: Ninja Singleton - Model Notation

For a program P and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Definition 40, the model notation of the Ninja Singleton mechanism $(P, E) \xleftarrow[NinjaSingleton]{represents} M$ is defined as follows:

The model M comprises one object of the class *class* of the meta model M_{Meta}.

$$object \in O_M, object \xrightarrow{instanceOf} class$$

The model notation defines no element as entry point.

$$E = \emptyset$$

Discussion The Ninja Singleton mechanism is usable when exactly one instance of the class must exist. The meta model class must not have any attribute. The class may have references to other classes. When one of these references is a containment reference, the targeted class must not be containable by other classes. When these requirements are met, the mechanism allows to have a single instance of the class, with all elements referenced by references being referenced by that single element. E.g. all components and interfaces can be contained by one single architecture element. This can be used to create root objects in models.

Namespace Hierarchy

The Namespace Hierarchy mechanism is usable when a tree of objects of the same class is to be represented. Using the Namespace Hierarchy mechanism, objects of a class, that has an identifying attribute and a containment reference to itself, are represented by a namespace hierarchy.

Example Figure 5.31 shows an example of the Namespace Hierarchy mechanism. The meta model specifies a class *Namespace* with the attribute *name*, a *String*, and a containment reference *children* to itself with the cardinality *0..**. The model instantiates this class with a hierarchy of objects with the names *org*, *codeling*, and *examples*. The corresponding code shows a corresponding namespace hierarchy including the namespaces identified by `org.codeling.examples`.

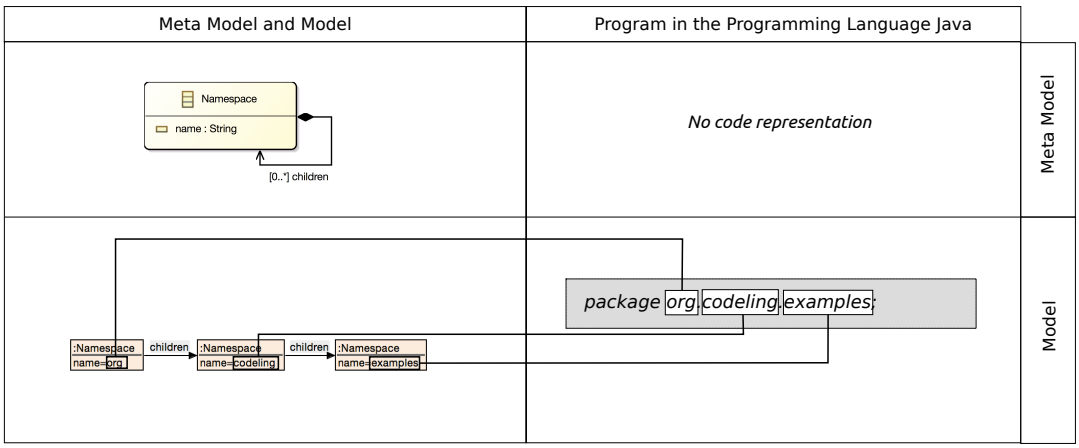


Figure 5.31: Example code for the Namespace Hierarchy Mechanism

Formalization The Namespace Hierarchy mechanism is formalized as follows:

Definition 42: Namespace Hierarchy - Meta Model Notation

The meta model notation of the Namespace Hierarchy mechanism $P \xleftrightarrow[NamespaceHierarchy]{represents}$ M_{Meta} is defined as follows:

The meta model M_{Meta} comprises a class, an attribute, and a containment reference.

$$class \in Classes_{M_{Meta}}, id \in Attributes_{M_{Meta}}, r \in Containments_{M_{Meta}}$$

The attributes and references of the meta model are defined as follows:

$$class.id \xrightarrow{isOfType} String,$$
$$class.r \xrightarrow{isOfType} class$$

Definition 43: Namespace Hierarchy - Model Notation

For a program P and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Definition 42, the model notation of the Namespace Hierarchy mechanism $(P, E) \xleftrightarrow[NamespaceHierarchy]{represents} M$ is defined as follows:

The model M comprises two objects of the class $class$ of the meta model M_{Meta}.

$$o^1, o^2 \in O_M; o^1, o^2 \xrightarrow{instanceOf} class$$

The parent object references the child object with the containment reference.

$$o^1.r \xrightarrow{references} o^2$$

The program declares two namespaces.

$$n^1, n^2 \in O_P; n^1, n^2 \xrightarrow{instanceOf} \mathcal{N}$$

The namespaces n^1, n^2 are named after the values of the model element's identifying attributes. The namespace that represents the owning class references the namespace that represents the owned class.

$$n^1.name \xrightarrow{hasValue} value(o^1.id),$$
$$n^2.name \xrightarrow{hasValue} value(o^2.id),$$
$$n^1.children \xrightarrow{references} n^2$$

The model notation defines the namespaces as entry points.

$$E = \{n^1, n^2\}$$

Discussion For this mechanism to be applicable, the class to represent must have an identifying attribute of the type *String*, to be mappable to the name of the code element in programming languages. At design time the namespace hierarchy can be extracted and is guaranteed to be a tree (by Constraint 9). The Namespace Hierarchy mechanism is only applicable for tree structures of identifiable elements. The class must have no attributes but one for its identification. It is therefore only applicable in special cases, e.g. a namespace concept. This mechanism can be used for a maximum of one class in a meta model, because no meta model code structures exist, that could be used to distinguish between two classes.

The entry points of the Namespace Hierarchy mechanism are the namespaces. It is possible to declare the namespaces' execution semantics with types, interfaces, or annotations that are declared within them. An execution runtime might respect the namespaces when mediating between providers and users of execution semantics within the code, e.g. between components of different layers. The use of namespace hierarchies is common in most modern programming languages.

5.6.3 Containment Representation

Integration mechanisms for containment representations can be used to represent containment references in a meta model and the assignment of targets in a model with program code structures. The applicability of integration mechanisms for containment references depends on the reference's owner to be represented with a specific mechanism. Table 5.3 shows integration mechanisms for representing containment references in models with program code structures, and their requirements regarding the notations for the owning classes.

Containment Operation for Types

This mechanism uses an operation signature and an operation for representing a reference, including the contained object. An annotation is used to mark the operation signature and operation a representation of the model reference. The Containment Operation Mechanism has two variants. This variant is used when the reference's owner class is translated with a type

Containment Reference Representations	
Name	**Owner**
Containment Operation for Types	Represented as type
Containment Operation for Interfaces	Represented as interface

Table 5.3: An overview of integration mechanisms for representing containment references with program code structures, and their requirements

declaration. The variant for interfaces (see Definitions 46 and 47) can be used when the owner class is translated with an interface declaration.

Example An example of this mechanism is shown in Figure 5.32. The meta model comprises a source class *State*, which owns a reference *transitions* to the class *Transitions*. The target class specifies an attribute *name*, a *String*. The model defines an object of the class *State* with the name *Ready*, which targets an object of the class *Transition* with the name *scanItem* with the reference.

The code specifies the annotation `Transitions` as a representation of the *transition* reference. The type `Ready` represents the source object of the class *State* with the Marker Interface mechanism. It owns an operation `scanItem` whose name equals the target object's name attribute value. The annotation `Transitions` is attached to the operation to mark it a representation of the contained object, including the reference.

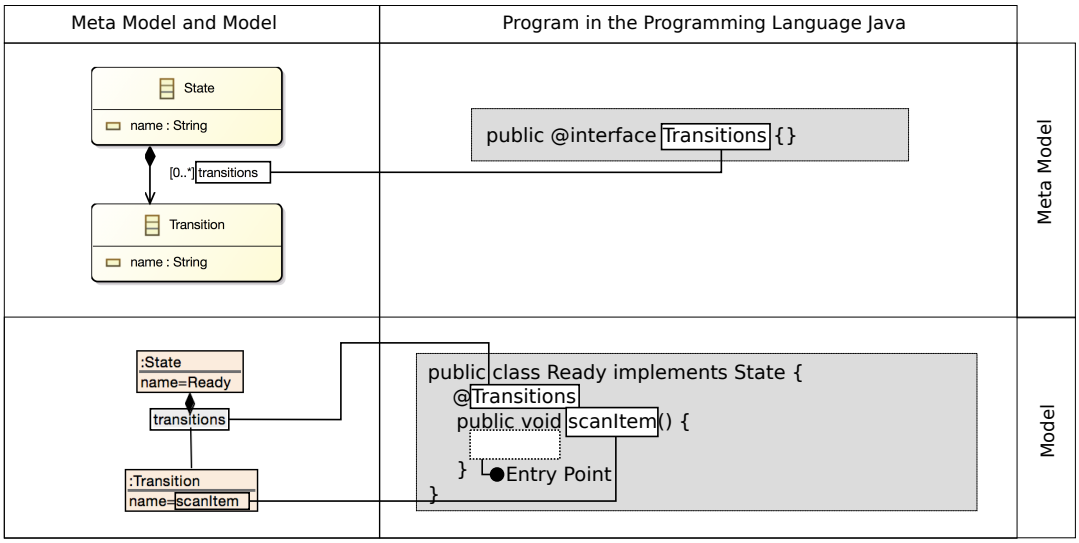


Figure 5.32: Example code for the Containment Operation for Types mechanism

Formalization The Containment Operation for Types mechanism is formalized as follows:

Definition 44: Containment Operation for Types - Meta Model Notation

The meta model notation of the Containment Operation for Types mechanism $P \xleftrightarrow[ContainmentOperationforTypes]{represents} M_{Meta}$ is defined as follows:
The meta model M_{Meta} defines a source class, a target class and a containment reference.

$$class_{source}, class_{target} \in Classes_{M_{Meta}}; reference \in Containments_{M_{Meta}}$$

The $reference$ is owned by $class_{source}$ and references $class_{target}$.

$$class_{source}.reference \xrightarrow{isOfType} class_{target}$$

The program declares an annotation.

$$annotation \in O_P, annotation \xrightarrow{instanceOf} \mathcal{A}$$

The annotation name equals the reference's name.

$$annotation.name \xrightarrow{hasValue} name(reference)$$

Definition 45: Containment Operation for Types - Model Notation

For a program P and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Definition 44, the model notation of the Containment Operation for Types mechanism $(P, E) \xleftrightarrow[ContainmentOperationforTypes]{represents} M$ is defined as follows:

Program structures $P_{lib}^{Reference}$ exist, that represent the meta model $M_{Meta}^{Reference}$ with the *Containment Operation for Types* meta model notation. A containment reference is notated therein with an annotation. The program references these program structures.

$$P_{lib}^{Reference} \xleftrightarrow[ContainmentOperationforTypes]{represents} M_{Meta}^{Reference}, reference \in References_{M_{Meta}^{Reference}},$$
$$annotation \in O_{P_{lib}^{Reference}}, annotation \xrightarrow{instanceOf} \mathcal{A}, P_{lib}^{Reference} \in R_P$$

Program structures P_{lib}^{source} exist, that represent the model M^{source} with a model notation that represents a source object with a type. The source object's class owns the reference. The program references these program structures.

$$P_{lib}^{source} \xleftrightarrow{represents} M^{source}, object_{source} \in O_{M^{source}},$$
$$object_{source} \xrightarrow{instanceOf} class_{source}, class_{source} \xrightarrow{has} reference,$$
$$type_{source} \in O_{P_{lib}^{source}}, type_{source} \xrightarrow{instanceOf} \mathcal{T}, P_{lib}^{source} \in R_P$$

The model defines a target object, an instance of the target class. The target class has an identifying attribute. The source object targets the target object with its reference.

$$object_{target} \in O_M, object_{target} \xrightarrow{instanceOf} class_{target},$$
$$class_{target}.id \xrightarrow{isOfType} String,$$
$$reference \xrightarrow{isOfType} class_{target},$$
$$(object_{source}, reference) \xrightarrow{references} object_{target}$$

The program declares an operation signature and a corresponding operation, that are owned by the source type.

$$os \in O_P, os \xrightarrow{instanceOf} \mathcal{OS}, \qquad type_{source}.operationSignature \xrightarrow{references} os,$$
$$o \in O_P, o \xrightarrow{instanceOf} \mathcal{O}, \qquad type_{source}.operations \xrightarrow{references} o,$$
$$o.signature \xrightarrow{references} os$$

The operation signature's name equals the value of the target object's identifying attribute. The annotation of the meta model notation is attached to the operation signature.

$$os.name \xrightarrow{hasValue} value(object.id) \qquad annotation \xrightarrow{attachedTo} os$$

The model notation defines the operation signature and the operation as entry point, meaning that the signature can be enhanced with further annotations, and the operation can describe arbitrary behaviour.

$$E = \{os, o\}$$

Discussion For the Containment Operation for Types mechanism to be applicable, the source object must be represented by a type and the target object must have an identifying attribute. If the source object references more than one target object with the reference, the model notation is applied multiple times. Then further operations with annotation attachments are added.

At design time, the reference and its target object can be identified by the annotation attachment. The mechanism does not allow for multiple contained objects that have the same name, because the operation signature's names must not be equal. The model notation enforces no specific return type or parameter list. When a new code fragment has to be created following this notation, a return type *Void* and an empty parameter list should be assumed. An execution runtime or arbitrary program code can invoke the containment operations to trigger the execution semantics of the contained object. A variant of the Containment Operation for Types mechanism was used by Balz in [Bal11, Section 4.1.2.2].

Containment Operation for Interfaces

This variant of the Containment Operation mechanism uses an operation signature for representing a reference, including the contained object. Again, an annotation is used to mark the operation signature a representation of the model reference. This mechanism is closely related to the Containment Operation for Types mechanism. It differs in the required translation of

the source class. Because the source class is represented with an interface here, as opposed to a type in the variant for types, only operation signatures, not operations, can be used.

The definition of the meta model notation of the Containment Operation for Interfaces mechanism equals the definition of the meta model notation of the Containment Operation mechanism for types. The definition of the model notation deviates slightly from the model notation of the corresponding model notation. The source object is represented by an interface instead of a type. Consequently, no operation is defined in the program code, but only an operation signature.

Example An example of this mechanism is shown in Figure 5.33. The meta model comprises a source class *Interface*, which owns a reference *operations* to the target class *Operation*. The model defines an object of the class *Interface* with the name *IBarcodeScanner*, which targets an object of the class *Operation* with the name *scan* and the reference *operations*.

The code specifies the annotation `Operations` as a representation of the *operations* reference. The interface `IBarcodeScanner` represents the source object with the Static Interface mechanism. It owns an operation signature named `scan`, equal to the target object's name attribute value. The annotation `Operations` is attached to the operation to mark it a representation of the reference and contained object.

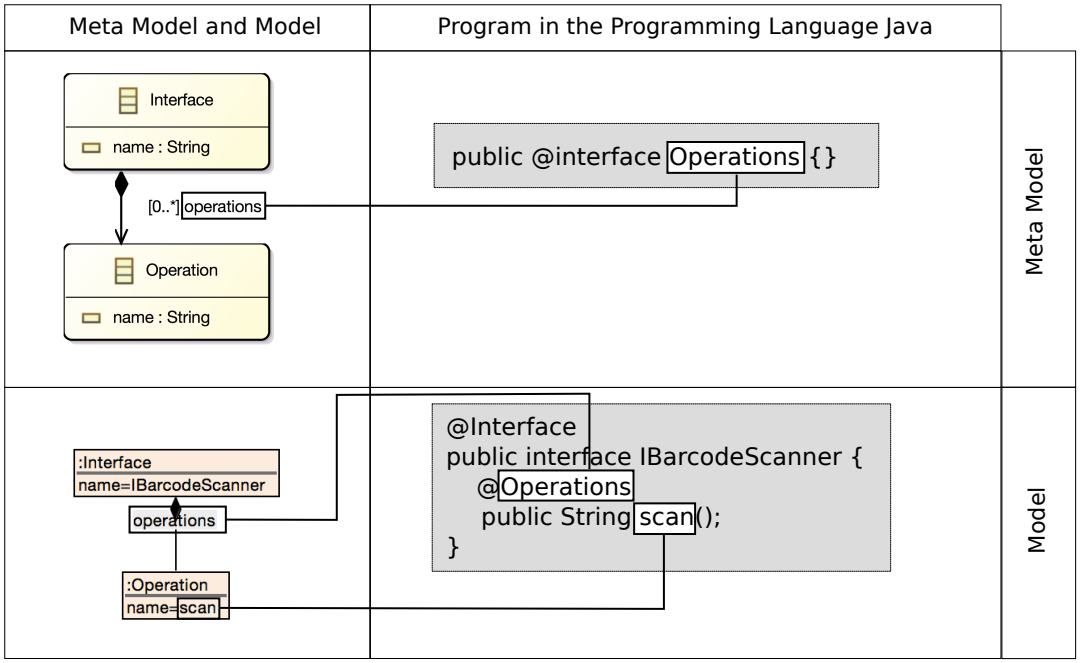


Figure 5.33: Example code for the Containment Operation for Interfaces mechanism

Formalization The Containment Operation for Interfaces mechanism is formalized as follows:

Definition 46: Containment Operation for Interfaces - Meta Model Notation

The meta model notation of the Containment Operation for Interfaces mechanism $P \xleftrightarrow[ContainmentOperationforInterfaces]{represents} M_{Meta}$ is defined as follows:

The meta model M_{Meta} defines a source class, a target class and a containment reference.

$$class_{source}, class_{target} \in Classes_{M_{Meta}}; reference \in Containments_{M_{Meta}}$$

The $reference$ is owned by $class_{source}$ and references $class_{target}$.

$$class_{source}.reference \xrightarrow{isOfType} class_{target}$$

The program declares an annotation.

$$annotation \in O_P, annotation \xrightarrow{instanceOf} \mathcal{A}$$

The annotation name equals the reference's name.

$$annotation.name \xrightarrow{hasValue} name(reference)$$

Definition 47: Containment Operation for Interfaces - Model Notation

For a program P and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Definition 44, the model notation of the Containment Operation for Interfaces mechanism $(P, E) \xleftrightarrow[ContainmentOperationforInterfaces]{represents} M$ is defined as follows:

Program structures $P_{lib}^{Reference}$ exist, that represent the meta model $M_{Meta}^{Reference}$ with the *Containment Operation for Interfaces* meta model notation. A containment reference is notated therein with an annotation. The program references these program structures.

$$P_{lib}^{Reference} \xleftrightarrow[ContainmentOperationforInterfaces]{represents} M_{Meta}^{Reference}, reference \in References_{M_{Meta}^{Reference}},$$
$$annotation \in O_{P_{lib}^{Reference}}, annotation \xrightarrow{instanceOf} \mathcal{A}, P_{lib}^{Reference} \in R_P$$

Program structures P_{lib}^{source} exist, that represent the model M^{source} with a model notation that represents a source object with an interface. The source object's class owns the reference. The program references these program structures.

$$P_{lib}^{source} \xleftrightarrow{represents} M^{source}, object_{source} \in O_{M^{source}},$$
$$object_{source} \xrightarrow{instanceOf} class_{source}, class_{source} \xrightarrow{has} reference,$$
$$interface_{source} \in O_{P_{lib}^{source}}, interface_{source} \xrightarrow{instanceOf} \mathcal{I}, P_{lib}^{source} \in R_P$$

The model defines a target object, an instance of the target class. The target class has an identifying attribute. The source object targets the target object with its reference.

$$object_{target} \in O_M, object_{target} \xrightarrow{instanceOf} class_{target},$$
$$class_{target}.id \xrightarrow{isOfType} String,$$
$$reference \xrightarrow{isOfType} class_{target},$$
$$(object_{source}, reference) \xrightarrow{references} object_{target}$$

The program declares an operation signature, that is owned by the source interface.

$$os \in O_P, os \xrightarrow{instanceOf} \mathcal{OS}, \qquad interface_{source}.operationSignature \xrightarrow{references} os$$

The operation signature's name equals the value of the target object's identifying attribute. The annotation of the meta model notation is attached to the operation signature.

$$os.name \xrightarrow{hasValue} value(object.id) \qquad annotation \xrightarrow{attachedTo} os$$

The model notation defines the operation signature as entry point, meaning that the signature can be enhanced with further annotations.

$$E = \{os\}$$

Discussion For the Containment Operation for Interfaces mechanism to be applicable, the source object must be represented by an interface and the target class must have an identifying attribute. If the source object references more than one target object with the reference, the model notation is applied multiple times. Then further operations with annotation attachments are added.

At design time, the reference, and its target object can be identified by the annotation. All implementers of the interface are required to implement the operation. The mechanism does not allow for multiple contained objects that have the same name, because the operation signature's names must not be equal. Implementers of the interface (e.g. via the Static Interface Implementation mechanism) are required to implement the operation. As in the variant for types, this variant enforces no specific return type or parameter list. When a new code fragment has to be created following this notation, a return type *Void* and an empty parameter list should be assumed. An execution runtime or arbitrary code can invoke the execution semantics of the implementers regarding the declared signature.

5.6.4 Reference Representation

Integration mechanisms for reference representations can be used to represent containment or non-containment references in a meta model and the assignment of targets in a model with program code structures. The applicability of integration mechanisms for references depend on the notation of the reference's owner and target. Table 5.4 shows integration mechanisms for representing these references with program code structures, and their dependencies.

Non-Containment Reference Representations		
Name	**Owner**	**Target**
Annotated Member Reference	Represented as type	Represented as type or interface
Static Interface Implementation	Represented as type	Static Interface Mechanism
Containment Operation Reference Annotation Parameter	Containment Operation	Represented as type or interface
Containment Operation Reference Parameter	Containment Operation	Represented as type or interface

Table 5.4: An overview of integration mechanisms for representing containment or non-containment references with program code structures, and their requirements

Annotated Member Reference to Type Annotation or Static Interface

The Annotated Member Reference mechanism is based on the idea to represent reference in the meta model and model as a member reference. An annotation is used to mark the member reference a representation of the model reference, and to declare the target class by its type. Based on the reference's cardinality and the representation of the target class, the mechanism has a set of variants. The following is the description for Reference Annotations to a target class that is translated with the Type Annotation or the Static Interface Mechanism. The other variants handle Reference Annotations that have target classes represented with the Marker Interface Mechanism with a cardinality *x..1* (Definitions 50 and 51) and *x..** (Definitions 52 and 53).

The definition of the meta model notation of the Annotated Member Reference to Type Annotation or Static Interface mechanism equals the definition of the meta model notation of the Containment Operation mechanisms.

Example An example of the variant with a Static Interface target is shown in Figure 5.34. The meta model comprises the source class *ComponentType*, which owns a reference *required* to the class *Interface*. The target class specifies an attribute *name*, a *String*. The model defines an object of the class *ComponentType* with the name *CashDesk*, which targets an object of the class *Interface* with the name *IBarcodeScanner* with the reference *required*.

The code specifies the annotation `Required` as a representation of the *required* reference. The annotation does not own an annotation parameter. The type `BarcodeScanner` represents the source object with the Type Annotation mechanism. It owns a member reference named `iBarcodeScanner`. The type of the member reference is the interface that represents the target object with the Static Interface mechanism. The annotation `Required` is attached to the reference to mark it a representation of the model reference.

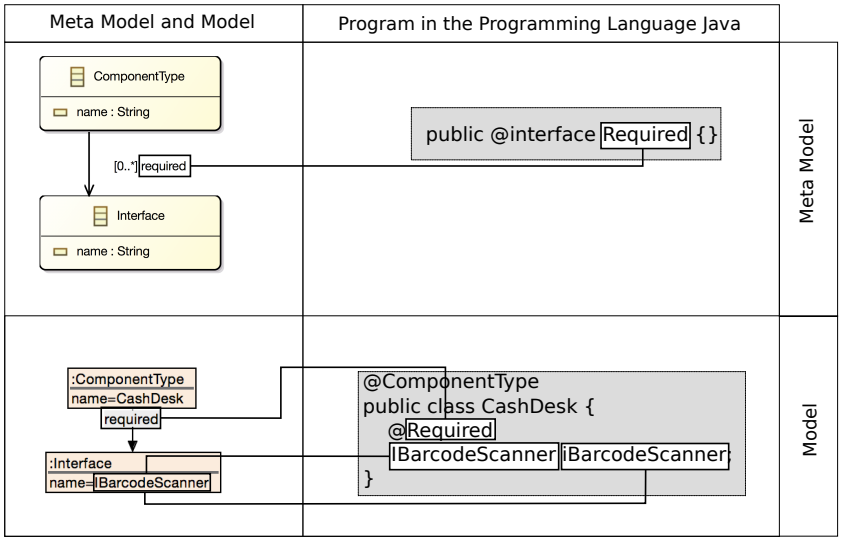


Figure 5.34: Example code for the mechanism Annotated Member Reference to Type Annotation or Static Interface

Formalization This variant of the Annotated Member Reference mechanism is formalized as follows:

Definition 48: Annotated Member Reference to Type Annotation or Static Interface - Meta Model Notation

The meta model notation of the Annotated Member Reference to Type Annotation or Static Interface mechanism $P \xleftrightarrow[AnnotatedMemberReference_{TA/SI}]{represents} M_{Meta}$ is defined as follows:

The meta model M_{Meta} defines a source class, a target class and a reference.

$$class_{source}, class_{target} \in Classes_{M_{Meta}}; reference \in References_{M_{Meta}}$$

The $reference$ is owned by $class_{source}$ and references $class_{target}$.

$$class_{source}.reference \xrightarrow{isOfType} class_{target}$$

The program declares an annotation.

$$annotation \in O_P, annotation \xrightarrow{instanceOf} \mathcal{A}$$

The annotation name equals the reference's name.

$$annotation.name \xrightarrow{hasValue} name(reference)$$

Definition 49: Annotated Member Reference to Type Annotation or Static Interface - Model Notation

For a program P and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Definition 48, the model notation of the Annotated Member Reference to Type Annotation or Static Interface mechanism $(P, E) \xleftarrow[AnnotatedMemberReference_{TA/SI}]{represents} M$ is defined as follows:

Program structures $P_{lib}^{Reference}$ exist, that represent the meta model $M_{Meta}^{Reference}$ with the Annotated Member Reference to Type Annotation or Static Interface meta model notation. The reference is notated therein with an annotation. The program references these program structures.

$$P_{lib}^{Reference} \xleftarrow[AnnotatedMemberReference_{TA/SI}]{represents} M_{Meta}^{Reference}, reference \in References_{M_{Meta}^{Reference}},$$
$$annotation \in O_{P_{lib}^{Reference}}, annotation \xrightarrow{instanceOf} \mathcal{A},$$
$$P_{lib}^{Reference} \in R_P$$

Program structures P_{lib}^{source} exist, that represent the model M^{source} with a model notation that represents a source object with a type. The program references these program structures.

$$P_{lib}^{source} \xleftarrow{represents} M^{source}, object_{source} \in O_{M^{source}},$$
$$object_{source} \xrightarrow{instanceOf} class_{source}, type_{source} \in O_{P_{lib}^{source}}, P_{lib}^{source} \in R_P$$

Program structures P_{lib}^{target} exist, that represent the model M^{target} with the Type Annotation mechanism or the Static Interface mechanism, representing a target object with a type or interface. The program references these program structures.

$$P_{lib}^{target} \xleftarrow[TypeAnnotation]{represents} M^{target} \vee P_{lib}^{target} \xleftarrow[StaticInterface]{represents} M^{target},$$
$$object_{target} \in O_{M^{target}}, object_{target} \xrightarrow{instanceOf} class_{target}, element_{target} \in O_{P_{lib}^{target}},$$
$$element_{target} \xrightarrow{instanceOf} \mathcal{T} \vee element_{target} \xrightarrow{instanceOf} \mathcal{I}, P_{lib}^{target} \in R_P$$

The reference is owned by the source class. The source object references the target object with the reference.

$$class_{source} \xrightarrow{has} reference, \qquad (object_{source}, reference) \xrightarrow{references} object_{target}$$

The program declares a member reference, that is owned by the source type.

$$mr \in O_P, mr \xrightarrow{instanceOf} \mathcal{MR},$$
$$type_{source}.memberReferences \xrightarrow{references} mr$$

The member reference's type is the type or interface that represents the target object. It is named after the target's identifying attribute value. The annotation is attached to the member reference to mark it a representation of the model reference.

$$mr \xrightarrow{isOfType} element_{target},$$
$$mr.name \xrightarrow{hasValue} value(object_{target}.id),$$
$$annotation \xrightarrow{attachedTo} mr$$

The model notation defines the member reference as entry point, meaning that it can be enhanced with further annotations.

$$E = \{mr\}$$

Discussion For the Annotated Member Reference to Type Annotation or Static Interface mechanism to be applicable, the source object must be represented by a type, the target class must be represented with the Type Annotation or the Static Interface mechanism. At design time, the annotation shows that the member reference represents the model reference's target. The target object can be identified based on the type name. This mechanism in all its variants allows arbitrary code to access the fields.

At runtime, an execution environment should inject an instance of the target type into the member reference as target. When the target is a Static Interface an execution runtime would need to identify a valid implementer of the static interface. Which instance to inject is subject to the execution environment, and depends on the semantics of the target class. Code in the entry point of the source type can then use the member reference's target. If operations have been inserted in the target's entry point, these can be called to call execution semantics of the target class.

This variant has some disadvantages. For each target of the reference a member reference is created, which may cause many member references that make the code hard to read. Also, when during a model change the targets are changed, the names of the member attributes change, and code relying on these names may be invalid. The concept behind this mechanism is known e.g. from CDI where CDI beans reference other CDI beans using an annotation *@Inject* [JSR14, Section 3.10].

Annotated Member Reference to Marker Interface for x..1 References

This variant of the Annotated Member Reference mechanism can be used when the target class is represented with code using the Marker Interface mechanism, and the reference has a cardinality of *0..1* or *1..1*. Instead of using the targeted type directly, the member reference is typed by the marker interface of the target class.

Example An example of this variant is shown in Figure 5.35. The meta model comprises a source class *StateMachine*, which owns a reference *initial* to the class *State*. The target class specifies an attribute *name*, a *String*. The model defines an object of the class *StateMachine* with the name *CashDesk*, which targets an object of the class *State* with the name *Ready* with the reference *initial*.

The code specifies the annotation `Initial` as a representation of the *initial* reference. The annotation owns a default parameter with the marker interface `State` as target type. The interface `State` (not shown) represents the target class with the Marker Interface mechanism. The type `CashDesk` represents the source object with the Marker Interface mechanism. It owns a member reference named `initial`. The type of the member reference is the marker interface `State` of the target class. The annotation `Initial` is attached to the reference to mark it a representation of the model reference. The type `Ready` is assigned to the default parameter to declare the referenced object.

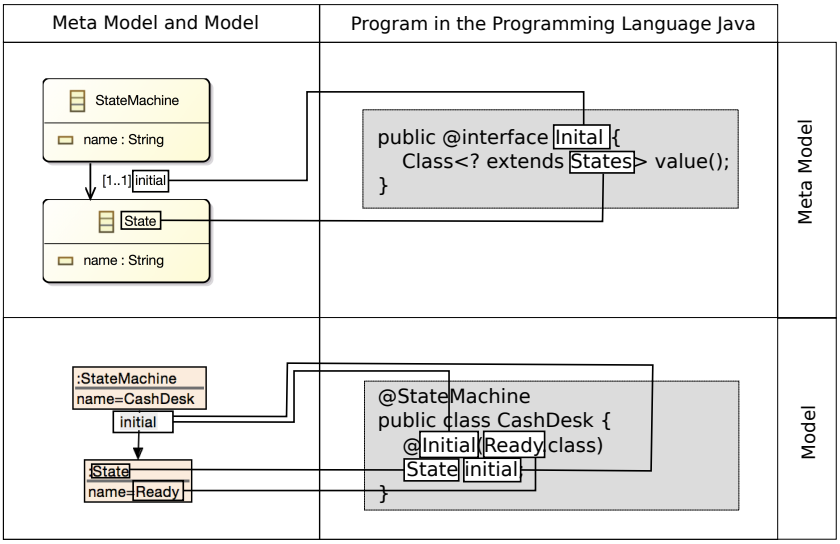


Figure 5.35: Example code for the Annotated Member Reference Mechanism to Marker Interface for x..1 References

Formalization This variant of the Annotated Member Reference mechanism is formalized as follows:

Definition 50: Annotated Member Reference to Marker Interface for x..1 References - Meta Model Notation

The meta model notation of the Annotated Member Reference to Marker Interface for x..1 References mechanism $P \xleftrightarrow[AnnotatedMemberReference_{MI}^{x..1}]{represents} M_{Meta}$ is defined as follows:

Program structures P_{lib}^{target} exist, that represent the meta model M_{Meta}^{target} with the Marker Interface meta model notation. The source class is notated therein with an interface. The program references these program structures.

$$P_{lib}^{target} \xleftrightarrow[MarkerInterface]{represents} M_{Meta}^{target}, class_{target} \in Classes_{M_{Meta}^{target}},$$

$$interface_{target} \in O_{P_{lib}^{target}}, annotation \xrightarrow{instanceOf} \mathcal{I}, P_{lib}^{target} \in R_P$$

The meta model M_{Meta} defines a source class and a reference. The reference has a cardinality of 0..1 or 1..1

$$class_{source} \in Classes_{M_{Meta}}, reference \in References_{M_{Meta}},$$
$$reference \xrightarrow{cardinality} 0..1 \oplus reference \xrightarrow{cardinality} 1..1$$

The $reference$ is owned by $class_{source}$ and references $class_{target}$.

$$class_{source}.reference \xrightarrow{isOfType} class_{target}$$

The program declares an annotation with an annotation parameter.

$$annotation \in O_P, annotation \xrightarrow{instanceOf} \mathcal{A},$$

$$ap \in O_P, ap \xrightarrow{instanceOf} \mathcal{AP}, annotation \xrightarrow{has} ap$$

The annotation name equals the reference's name. The annotation parameter's name equals the name of the target class. Its type is the interface that represents the target class.

$$annotation.name \xrightarrow{hasValue} name(reference),$$
$$ap.name \xrightarrow{hasValue} name(class_{target}),$$
$$ap.type \xrightarrow{references} interface_{target}$$

Definition 51: Annotated Member Reference to Marker Interface for x..1 References - Model Notation

For a program P and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Definition 50, the model notation of the Annotated Member Reference to Marker Interface for x..1 References mechanism $(P, E) \xleftarrow[AnnotatedMemberReference_{MI}^{x..1}]{represents} M$ is defined as follows:

Program structures $P_{lib}^{Reference}$ exist, that represent the meta model $M_{Meta}^{Reference}$ with the Annotated Member Reference to Marker Interface for x..1 References meta model notation for marker interfaces and x..1 cardinalities. The reference is notated therein with an annotation. The program references these program structures.

$$P_{lib}^{Reference} \xleftarrow[AnnotatedMemberReference_{MI}^{x..1}]{represents} M_{Meta}^{Reference}, reference \in References_{M_{Meta}^{Reference}},$$
$$annotation \in O_{P_{lib}^{Reference}}, annotation \xrightarrow{instanceOf} \mathcal{A},$$
$$P_{lib}^{Reference} \in R_P$$

Program structures P_{lib}^{source} exist, that represent the model M^{source} with a model notation that represents a source object with a type. The program references these program

structures.

$$P_{lib}^{source} \xleftrightarrow{represents} M^{source}, object_{source} \in O_{M^{source}},$$
$$object_{source} \xrightarrow{instanceOf} class_{source}, type_{source} \in O_{P_{lib}^{source}}, P_{lib}^{source} \in R_P$$

Program structures P_{lib}^{target} exist, that represent the model M^{target} with the Marker Interface mechanism, representing a target object with a type that implements a marker interface. The marker interface represents the reference's target class. The program references these program structures.

$$P_{lib}^{target} \xleftarrow[MarkerInterface]{represents} M^{target}, object_{target} \in O_{M^{target}}, object_{target} \xrightarrow{instanceOf} class_{target},$$
$$interface_{target} \in O_{P_{lib}^{target}}, interface_{target} \xrightarrow{instanceOf} \mathcal{I},$$
$$type_{target} \in O_{P_{lib}^{target}}, type_{target} \xrightarrow{instanceOf} \mathcal{T}, P_{lib}^{target} \in R_P$$

The reference is owned by the source class. The source object references the target object with the reference.

$$class_{source} \xrightarrow{has} reference, \qquad (object_{source}, reference) \xrightarrow{references} object_{target}$$

The program declares a member reference, that is owned by the source type.

$$mr \in O_P, mr \xrightarrow{instanceOf} \mathcal{MR},$$
$$type_{source}.memberReferences \xrightarrow{references} mr$$

The member reference's type is the interface that represents the target object. It is named after the reference's name. The annotation is attached to the member reference to mark it a representation of the model reference. The parameter value is set to the targeted element.

$$mr \xrightarrow{isOfType} interface_{target}, \qquad mr.name \xrightarrow{hasValue} name(r),$$
$$annotation \xrightarrow{attachedTo} mr, \qquad ap \xrightarrow{hasValue} type_{target}$$

The model notation defines the member reference as entry point, meaning that it can be enhanced with further annotations.

$$E = \{mr\}$$

Discussion For the Annotated Member Reference to Marker Interface for x..1 References mechanism to be applicable, the source object must be represented by a type, the target class must be represented with the Marker Interface mechanism, and the reference's cardinality must be *0..1* or *1..1*.

In contrast to the variant for Type Annotations and Static Interfaces, the name of the reference stays the same, when the targets in the model are changed. Arbitrary other code, that uses these references, can therefore rely on the member reference's name. In contrast to

the Containment Operation mechanism, this variant of this mechanism allows to reference the same target multiple times using different references, due to the use of the reference name as member reference name.

At run time, an execution environment should inject an instance of the target type into the member reference as target. Which instance to inject is subject to the execution environment, and depends on the semantics of the target class. Code in the entry point of the source type can then use the member reference' target. If operations have been inserted in the marker interface's entry point, these can be called to call execution semantics of the target class. The concept behind this mechanism is e.g. known from EJB, where Enterprise Beans reference other Enterprise Beans via their interfaces [EJB13, Section 3.4.1].

Annotated Member Reference to Marker Interface for x..* References

This variant is close to the variant to marker interfaces for *x..1* references. To respect the cardinality, the type of the member reference is an array of marker interfaces of the target class.

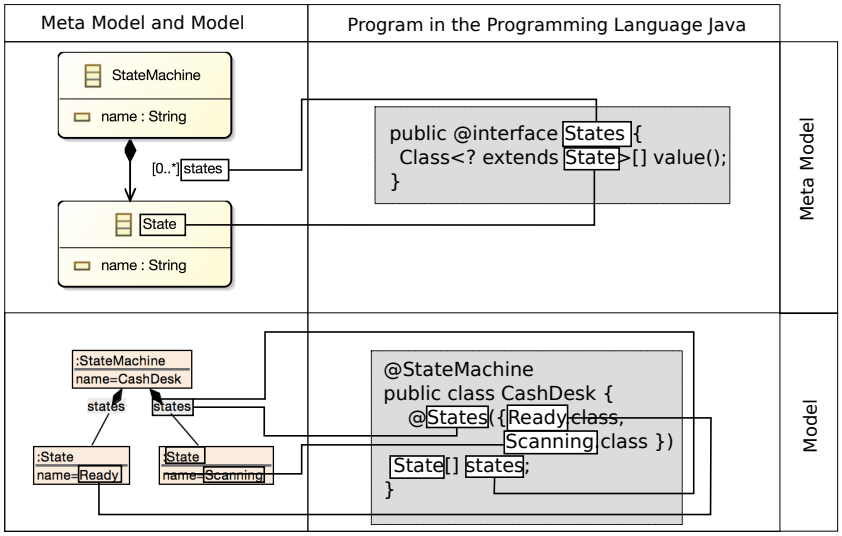


Figure 5.36: Example code for the Annotated Member Reference Mechanism to Marker Interface for x..* References

Example An example of this variant is shown in Figure 5.36. The meta model comprises a source class *StateMachine*, which owns a reference *states* to the class *State*. The reference has a cardinality of *0..**. The target class specifies an attribute *name*, a *String*. The model defines an object of the class *StateMachine* with the name *CashDesk*, which targets two objects of the class *State* with the names *Ready* and *Scanning* with the reference *states*[6].

The code specifies the annotation *States* as a representation of the *states* reference. It owns a default parameter with an array of the marker interface *State* as target type.

[6] In the running example the state machine references four states. This example has been shortened for a better readability.

The types *Ready* and *Scanning* (not shown) represent the target objects with the Marker Interface mechanism. The type *CashDesk* represents the source object with the Type Annotation mechanism. It owns a member reference named *states*. The type of the member reference is an array of the marker interface *State* of the target class. The annotation *States* is attached to the reference to mark it a representation of the model reference. The types *Ready* and *Scanning* are assigned to the default parameter to declare the referenced objects.

Formalization In the following, only the variation of this variant's formalization from the Definitions 50 and 51 is given.

Definition 52: Annotated Member Reference to Marker Interface for x..* References - Meta Model Notation

The meta model notation of the Annotated Member Reference to Marker Interface for x..* References mechanism $P \xleftrightarrow[AnnotatedMemberReference^{x..*}_{MI}]{represents} M_{Meta}$ is defined as follows:

Program structures P^{target}_{lib} exist, that represent the meta model M^{target}_{Meta} with the Marker Interface meta model notation. The source class is notated therein with an interface. The program references these program structures.

$$P^{target}_{lib} \xleftrightarrow[MarkerInterface]{represents} M^{target}_{Meta}, class_{target} \in Classes_{M^{target}_{Meta}},$$
$$interface_{target} \in O_{P^{target}_{lib}}, annotation \xrightarrow{instanceOf} \mathcal{I},$$
$$P^{target}_{lib} \in R_P$$

The meta model M_{Meta} defines a source class and a reference.

$$class_{source} \in Classes_{M_{Meta}}, reference \in References_{M_{Meta}}$$

The $reference$ is owned by $class_{source}$ and references $class_{target}$. It has a cardinality of $0..*$ or $1..*$

$$class_{source} \in Classes_{M_{Meta}}, reference \in References_{M_{Meta}},$$
$$reference \xrightarrow{cardinality} 0..* \oplus reference \xrightarrow{cardinality} 1..*$$

The program declares an annotation with an annotation parameter.

$$annotation \in O_P, annotation \xrightarrow{instanceOf} \mathcal{A},$$

$$ap \in O_P, ap \xrightarrow{instanceOf} \mathcal{AP}, annotation \xrightarrow{has} ap$$

The annotation name equals the reference's name. The annotation parameter's name equals the name of the target class. Its type is the interface that represents the target class. The type is an array.

$$annotation.name \xrightarrow{hasValue} name(reference), \quad ap.name \xrightarrow{hasValue} name(class_{target}),$$
$$ap.type \xrightarrow{references} interface_{target}, \qquad ap.isArrayType \xrightarrow{hasValue} true$$

Definition 53: Annotated Member Reference to Marker Interface for x..* References - Model Notation

For a program P and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Definition 50, the model notation of the Annotated Member Reference to Marker Interface for x..* References mechanism $(P, E) \xleftrightarrow[AnnotatedMemberReference_{MI}^{x..*}]{represents} M$ is defined as follows:

Program structures $P_{lib}^{Reference}$ exist, that represent the meta model $M_{Meta}^{Reference}$ with the Annotated Member Reference to Marker Interface for x..* References meta model notation for marker interfaces with x..* cardinalities. The reference is notated therein with an annotation. The program references these program structures.

$$P_{lib}^{Reference} \xleftrightarrow[AnnotatedMemberReference_{MI}^{x..*}]{represents} M_{Meta}^{Reference}, reference \in References_{M_{Meta}^{Reference}},$$
$$annotation \in O_{P_{lib}^{Reference}}, annotation \xrightarrow{instanceOf} \mathcal{A},$$
$$P_{lib}^{Reference} \in R_P$$

Program structures P_{lib}^{source} exist, that represent the model M^{source} with a model notation that represents a source object with a type. The program references these program structures.

$$P_{lib}^{source} \xleftrightarrow{represents} M^{source}, object_{source} \in O_{M^{source}},$$
$$object_{source} \xrightarrow{instanceOf} class_{source}, type_{source} \in O_{P_{lib}^{source}}, P_{lib}^{source} \in R_P$$

Program structures P_{lib}^{target} exist, that represent the model M^{target} with the Marker Interface mechanism, representing a target object with a type that implements a marker interface. The marker interface represents the reference's target class. The program references these program structures.

$$P_{lib}^{target} \xleftrightarrow[MarkerInterface]{represents} M^{target}, object_{target} \in O_{M^{target}}, object_{target} \xrightarrow{instanceOf} class_{target},$$
$$interface_{target} \in O_{P_{lib}^{target}}, interface_{target} \xrightarrow{instanceOf} \mathcal{I},$$
$$type_{target} \in O_{P_{lib}^{target}}, type_{target} \xrightarrow{instanceOf} \mathcal{T}, P_{lib}^{target} \in R_P$$

The reference is owned by the source class. The source object references the target object with the reference. The reference has a cardinality of 0..1 or 1..1

$$class_{source} \xrightarrow{has} reference, (object_{source}, reference) \xrightarrow{references} object_{target}$$
$$reference \xrightarrow{cardinality} 0..1 \oplus reference \xrightarrow{cardinality} 1..1$$

The program declares a member reference, that is owned by the source type.

$$mr \in O_P, mr \xrightarrow{instanceOf} \mathcal{MR},$$
$$type_{source}.memberReferences \xrightarrow{references} mr$$

The member reference's type is the interface that represents the target object. It is an array type, named after the reference's name. The annotation is attached to the member reference to mark it a representation of the model reference. The parameter value is set to the targeted element.

$$mr \xrightarrow{isOfType} interface_{target}, \qquad mr.name \xrightarrow{hasValue} name(r),$$
$$mr.isArrayType \xrightarrow{hasValue} true, \qquad annotation \xrightarrow{attachedTo} mr,$$
$$ap \xrightarrow{hasValue} type_{target}$$

The model notation defines the member reference as entry point, meaning that it can be enhanced with further annotations.

$$E = \{mr\}$$

Discussion For the Annotated Member Reference to Marker Interface for x..* References mechanism to be applicable, the source object must be represented by a type, the target class must be represented with the Marker Interface mechanism, and the reference's cardinality must be *0..** or *1..**. This variant of this mechanism allows to reference the same target multiple times using different references, due to the use of the reference name as member reference name.

At design time, the reference's targets can be identified by the annotation parameter value. In contrast to the Containment Operation mechanism, this variant of this mechanism allows for referencing the same target multiple times using different references, due to the use of the reference name as member reference name. An empty reference can be represented with an empty array in the annotation parameter. Arbitrary code can therefore rely on the existence, type, and name of the member reference. Apart from that, this variant of the Annotated Member Reference mechanism has the same properties that are discussed for the variant for *x..1* cardinalities.

Static Interface Implementation

The Static Interface Implementation mechanism requires the source class to be represented with a type. When the target class is represented with the Static Interface mechanism, the type that represents the source class can implement the interface to add a representation for the reference.

Example An example of this mechanism is shown in Figure 5.37. The meta model defines a source class *ComponentType*, which owns a reference *provided* to the class *Interface*. The target class specifies an attribute *name*, a *String*. The model defines an object of the class *ComponentType* with the name *BarcodeScanner*, which targets an object of the class *Interface* with the name *IBarcodeScanner* with the reference *provided*.

The code specifies the type `BarcodeScanner`, which represents the source object with the Type Annotation mechanism. It implements the interface `IBarcodeScanner`, which represents the target object with the Static Interface mechanism.

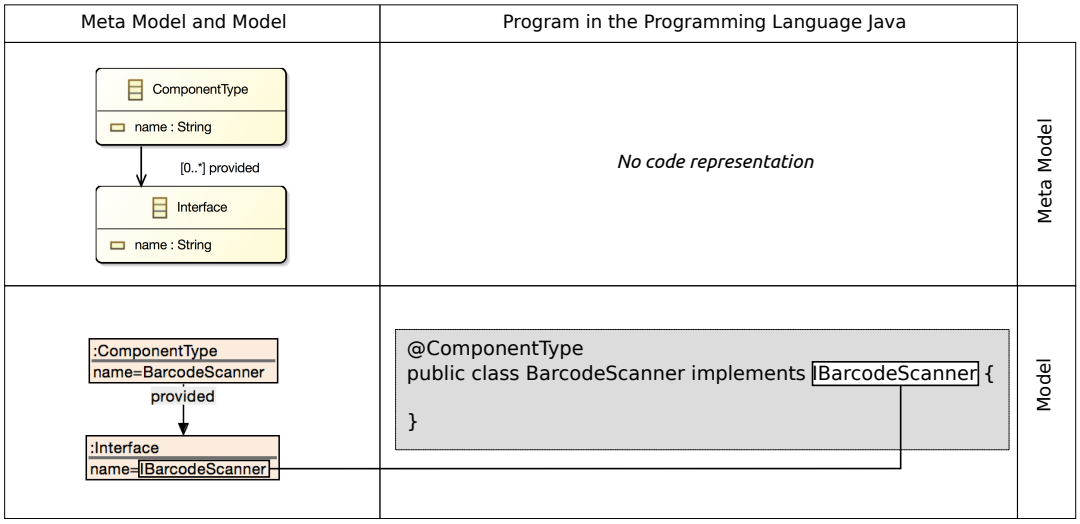


Figure 5.37: Example code for the Static Interface Implementation mechanism

Formalization The Static Interface Implementation mechanism to is formalized as follows:

Definition 54: Static Interface Implementation - Meta Model Notation

The meta model notation of the Static Interface Implementation mechanism $P \xleftrightarrow[StaticInterfaceImplementation]{represents} M_{Meta}$ is defined as follows:

The meta model M_{Meta} comprises a source class, a target class, and a reference between them.

$$class_{source}, class_{target} \in Classes_{M_{Meta}},$$

$$reference \in References_{M_{Meta}}$$

The source class owns the reference. The reference targets the target class.

$$class_{source}.reference \xrightarrow{isOfType} class_{target}$$

Definition 55: Static Interface Implementation - Model Notation

For a program P and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Definition 54, the model notation of the Static Interface Implementation mechanism $(P, E) \xleftarrow[StaticInterfaceImplementation]{represents} M$ is defined as follows:

Program structures P_{lib}^{source} exist, that represent the model M^{source} with a model notation that represents a source object with a type. The program references these program structures.

$$P_{lib}^{source} \xleftarrow{represents} M^{source}, object_{source} \in O_{M^{source}}, object_{source} \xrightarrow{instanceOf} class_{source},$$
$$type_{source} \in O_{P_{lib}^{source}}, type_{source} \xrightarrow{instanceOf} \mathcal{T}, P_{lib}^{source} \in R_P$$

Program structures P_{lib}^{target} exist, that represent the model M^{target} with the Static Interface mechanism, representing an object $object_{target} \xrightarrow{instanceOf} class_{target}$ with an interface $interface$. The program P references these program structures.

$$P_{lib}^{target} \xleftarrow[StaticInterface]{represents} M^{target}, object_{target} \in O_{M_{target}}, interface \in O_{P_{lib}^{target}},$$
$$interface \xrightarrow{instanceOf} \mathcal{I}, P_{lib}^{target} \in R_P$$

The source object references the target object with the reference.

$$(object_{source}, reference) \xrightarrow{references} object_{target}$$

The source type implements the target interface.

$$type_{source} \xrightarrow{implements} interface$$

The model notation does not define an entry point.

$$E = \emptyset$$

Discussion For the Static Interface Implementation mechanism to be applicable, the source object must be represented by a type, the target class must be represented with the Static Interface mechanism. At design time, the target class can be identified by the annotation attached to the interface in the Static Interface mechanism. All operations of the static interface need to be implemented by the type. Types that implement a static interface can be instantiated by an execution runtime and injected e.g. into member references. This mechanism can be used

to provide execution semantics declared with the Static Interface mechanism (see Definitions 38 and 39).

Containment Operation Reference Annotation Parameter to Type Annotation or Static Interface for x..1 References

The mechanisms for references above rely on the target being represented as a type or interface. When a class is represented with an operation, as it is performed when using the Containment Operation mechanism, these mechanisms are not applicable. The Containment Operation Reference Annotation Parameter mechanism allows for specifying targets for references of these classes, by leveraging the annotation introduced by the Containment Operation mechanism. The mechanism has four variants. Here, the variant for references with a cardinality of *0..1* or *1..1* and targets represented with the Type Annotation or the Static Interface mechanism is shown. Following the description of this variant, the other variants are for the same type of targets with reference cardinalities of *0..** or *1..**, and with targets represented with the Marker Interface mechanism.

Example An example of the Containment Operation Reference Annotation Parameter to Type Annotation or Static Interface for x..1 References is shown in Figure 5.38. The meta model comprises a source class *Operation*, which owns a reference *executionStrategy* to the class *ExecutionStrategy*. The target class specifies an attribute *name*, a *String*. The model defines an object of the class *Operation* with the name *scan*, which targets an object of the class *ExecutionStrategy* with the name *ExecuteLocally* using the reference *executionStrategy*. This execution strategy declares that the operation will be executed on the local processor. The alternative would be *ExecuteInCloud* to execute a long running task in a cloud context for a better performance.

The code shows the Containment Operation Annotation that represents the containment reference with the type *Operation*. This mechanism extends the annotation with an annotation parameter named after the reference name. The parameter references any type.

For the model notation, the code shows a containment operation named `scan`. The annotation parameter declared in the meta model notation is set to the type `ExecuteLocally`, which represents the corresponding strategy object with the Type Annotation mechanism.

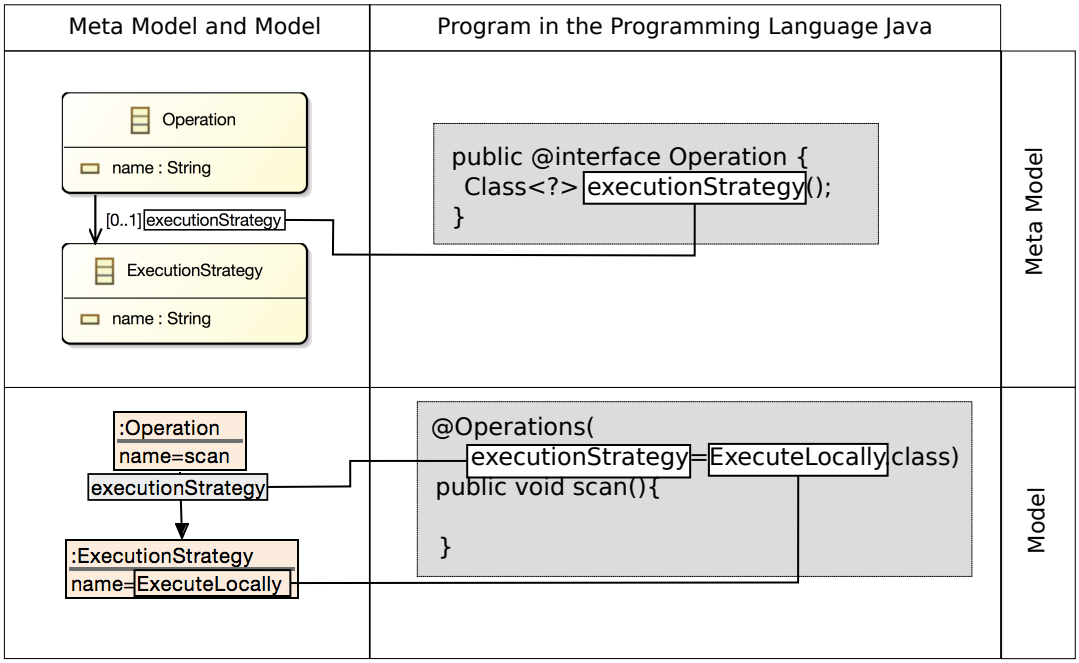


Figure 5.38: Example code for the mechanism Containment Operation Reference Annotation Parameter to Type Annotation or Static Interface for x..1 References

Formalization This variant of the Containment Operation Reference Annotation Parameter mechanism is formalized as follows:

Definition 56: Containment Operation Reference Annotation Parameter to Type Annotation or Static Interface for x..1 References - Meta Model Notation

The meta model notation of the Containment Operation Reference Annotation Parameter mechanism $P \overset{represents}{\Longleftrightarrow}_{ContainmentOperationReferenceAnnotationParameter^{x..1}_{TA/SI}} M_{Meta}$ is defined as follows:

Program structures P^{target}_{lib} exist, that represent the meta model M^{target}_{Meta} with either the $\overset{represents}{\Longleftrightarrow}_{TypeAnnotation}$ or the $\overset{represents}{\Longleftrightarrow}_{StaticInterface}$ meta model notation, which represents a target class. The program references these program structures.

$$P^{target}_{lib} \overset{represents}{\Longleftrightarrow}_{TypeAnnotation} M^{target}_{Meta} \oplus P^{target}_{lib} \overset{represents}{\Longleftrightarrow}_{StaticInterface} M^{target}_{Meta},$$
$$class_{target} \in Classes_{M^{target}_{Meta}}, P^{target}_{lib} \in R_P$$

The meta model M_{Meta} defines a source class and a reference.

$$class_{source} \in Classes_{M_{Meta}},$$
$$reference \in References_{M_{Meta}},$$

The reference is owned by the source class and references the target class. The reference has a cardinality of 0..1 or 1..1.

$$class_{source}.reference \xrightarrow{isOfType} class_{target},$$
$$class_{source}.reference \xrightarrow{cardinality} 0..1 \oplus class_{source}.reference \xrightarrow{cardinality} 1..1$$

The program declares an annotation with an annotation parameter.

$$annotation \in O_P, annotation \xrightarrow{instanceOf} \mathcal{A},$$

$$ap \in O_P, ap \xrightarrow{instanceOf} \mathcal{AP}, annotation \xrightarrow{has} ap$$

The annotation name equals the reference's name. The annotation parameter's name equals the name of the reference. It declares not type, so that each type declaration can be referenced by the parameter.

$$annotation.name \xrightarrow{hasValue} name(reference),$$
$$ap.name \xrightarrow{hasValue} name(class_{target})$$

Definition 57: Containment Operation Reference Annotation Parameter to Type Annotation or Static Interface for x..1 References - Model Notation

For a program P and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Definition 56, the model notation of the Containment Operation Reference Annotation Parameter to Type Annotation or Static Interface for x..1 References mechanism $(P, E) \xleftarrow[ContainmentOperationReferenceAnnotationParameter^{x..1}_{TA/SI}]{represents} M$ is defined as follows:

Program structures $P^{Reference}_{lib}$ exist, that represent the meta model $M^{Reference}_{Meta}$ with the meta model notation $\xleftrightarrow[ContainmentOperationReferenceAnnotationParameter^{x..1}_{TA/SI}]{represents}$. A reference is notated therein with an annotation parameter. The corresponding annotation is attached to the operation signature. The program references these program structures.

$$P^{Reference}_{lib} \xleftrightarrow[ContainmentOperationReferenceAnnotationParameter^{x..1}_{TA/SI}]{represents} M^{Reference}_{Meta},$$
$$reference \in References_{M^{Reference}_{Meta}}; annotation, ap \in O_{P^{Reference}_{lib}};$$
$$annotation \xrightarrow{instanceOf} \mathcal{A}; ap \xrightarrow{instanceOf} \mathcal{AP};$$

$$annotation.parameters \xrightarrow{references} ap; annotation \xrightarrow{attachedTo} os; P_{lib}^{Reference} \in R_P$$

Program structures P_{lib}^{source} exist, that represent the model M^{source} with either the $\xleftarrow[ContainmentOperationforTypes]{represents}$ or the $\xleftarrow[ContainmentOperationforInterfaces]{represents}$ model notation, which represent an source object with an operation signature. The program references these program structures.

$$P_{lib}^{source} \xleftarrow[ContainmentOperationforTypes]{represents} \oplus \xleftarrow[ContainmentOperationforInterfaces]{represents} M^{source},$$
$$o_{source} \in O_{M^{source}}, os \in O_{P_{lib}^{source}}, os \xrightarrow{instanceOf} \mathcal{OS}, P_{lib}^{source} \in R_P$$

Program structures P_{lib}^{target} exist, that represent the model M^{target} with the Type Annotation mechanism or the Static Interface mechanism, representing a target object with a type or interface. The program references these program structures.

$$P_{lib}^{target} \xleftarrow[TypeAnnotation]{represents} M^{target} \oplus P_{lib}^{target} \xleftarrow[StaticInterface]{represents} M^{target},$$
$$o_{target} \in O_{M^{target}}, o_{target} \xrightarrow{instanceOf} class_{target}, type_{target} \in O_{P_{lib}^{target}},$$
$$e_{target} \xrightarrow{instanceOf} \mathcal{T} \oplus e_{target} \xrightarrow{instanceOf} \mathcal{I}, P_{lib}^{target} \in R_P$$

The source object references the target object with the reference.

$$(o_{source}, reference) \xrightarrow{references} o_{target}$$

The annotation is attached to the operation signature, and its parameter is set to the target type or interface.

$$(os, annotation, ap) \xrightarrow{hasValue} e_{target}$$

The model notation defines no entry point

$$E = \emptyset$$

Discussion For the Containment Operation Reference Annotation Parameter to Type Annotation or Static Interface for x..1 References mechanism to be applicable, the source object must be represented with the Containment Operation mechanism, the target class must be represented with the Type Annotation or the Static Interface mechanism, and the reference's cardinality must be *0..1* or *1..1*.

During design time, the target is defined by the annotation parameter value. But the assignment of targets within the code is not type safe. As the targeted types or interfaces do not have a common denominator such as types implementing a marker interface, the compiler will not prevent a developer to enter an invalid type or interface in the program code as target.

When the reference has a cardinality of *0..1*, its reference might have no target assigned to it. In that case, no value would be assigned to the parameter of the attached annotation, which contradicts Definition 31. In that case a type (or interface) has to be declared that represents

the empty target. This might be counterintuitive. Java, e.g. allows for setting default values for annotation parameters. This concept could be used to set such a type or interface as default target, which is known to a translation or execution runtime.

An execution runtime can access the targeted type or interface using introspection mechanisms, and use instances of the type e.g. to evaluate whether the Containment Operation should be invoked. The program code within the operation can access the value using the same introspection mechanisms, but no instance of the target type or interface is made available directly, e.g. via an operation parameter. This would not be possible for targets translated with the Type Annotation or the Static Interface mechanism with a cardinality of x..1. Such parameters would be named after the reference or after the target type's or interface's name. None of these might be unambiguous within the list of parameters, but parameters must have distinguishable names. This is a variant of a code structure used by Balz, e.g. for the target of transitions in state machines [Bal11, Section 4.1.2.2].

Containment Operation Reference Annotation Parameter to Type Annotation or Static Interface for x..* References

This variant is for references with a cardinality of *0..** or *1..** and targets translated with the Type Annotation or the Static Interface mechanism. In contrast to the variant for *0..1* or *1..1* cardinalities, this variant uses an annotation parameter with an array type to represent multiple targets.

Example An example of the Containment Operation Reference Annotation Parameter to Type Annotation or Static Interface for x..* References is shown in Figure 5.39. The meta model comprises a source class *Operation*, which owns a reference *rolesAllowed* to the class *Role*. The target class specifies an attribute *name*, a *String*. The model defines an object of the class *Operation* with the name *scan*, which targets one object with the name *Cashier*, and one object with the name *StoreManager* of the class *Role* with the reference *rolesAllowed*. The reference declares that users in the respective roles are allowed to invoke the operation.

The code shows the Containment Operation Annotation that represents the containment reference towards the class *Operation*. This mechanism extends the annotation with an annotation parameter named after the reference name. The parameter references an array of any type. For the model notation, the code shows a containment operation named `scan`. The annotation parameter declared in the meta model notation is set to an array that contains the `Cashier` and the `StoreManager` type, which represent the corresponding roles with the Type Annotation mechanism.

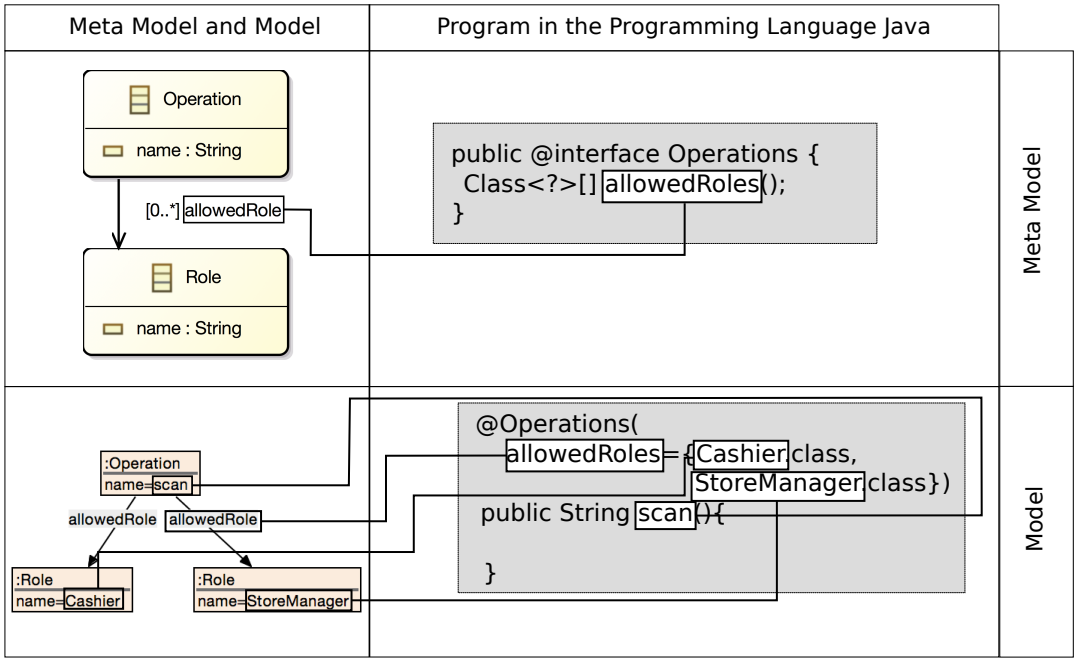


Figure 5.39: Example code for the mechanism Containment Operation Reference Annotation Parameter to Type Annotation or Static Interface for x..* References

Formalization In the following, only the deviation of this variant's formalization from Definitions 56 and 57 is given.

Definition 58: Containment Operation Reference Annotation Parameter to Type Annotation or Static Interface for x..* References - Meta Model Notation

The meta model notation of the Containment Operation Reference Annotation Parameter mechanism $P \xleftrightarrow{represents}{}_{ContainmentOperationReferenceAnnotationParameter^{x..*}_{TA/SI}} M_{Meta}$ deviates from the definition of $\xleftrightarrow{represents}{}_{ContainmentOperationReferenceAnnotationParameter^{x..1}_{TA/SI}}$ as follows: follows:

Instead of a cardinality of either *0..1* or *1..1*, the reference has a cardinality of either *0..** or *1..**.

$$class_{source}.reference \xrightarrow{cardinality} 0..* \oplus class_{source}.reference \xrightarrow{cardinality} 1..*$$

The annotation parameter declared in the program is of an array type.

$$ap.isArrayType \xrightarrow{hasValue} true$$

Definition 59: Containment Operation Reference Annotation Parameter to Type Annotation or Static Interface for x..* References - Model Notation

The model notation of the Containment Operation Reference Annotation Parameter mechanism $P \xleftarrow[ContainmentOperationReferenceAnnotationParameter^{x..*}_{TA/SI}]{represents} M_{Meta}$ varies from the definition of $\xleftarrow[ContainmentOperationReferenceAnnotationParameter^{x..1}_{TA/SI}]{represents}$ as follows: follows:

Instead of the meta model notation for *0..1* or *1..1* references, this variant uses the meta model notation for *0..** or *1..** references.

$$P^{Reference}_{lib} \xleftrightarrow[ContainmentOperationReferenceAnnotationParameter^{x..*}_{TA/SI}]{represents} M^{Reference}_{Meta}$$

Discussion For this variant to be applicable, the source object must be represented with the Containment Operation mechanism, the target class must be represented with the Type Annotation or the Static Interface mechanism, and the reference's cardinality must be *0..** or *1..**.

As in the variant for *0..1* or *1..1* cardinalities, the target is defined by the annotation parameter value at design time. The assignment of targets within the code is also not type safe. When the reference has a cardinality of *0..**, its reference might have no target assigned to it. In that case, an empty array can be given as parameter value. The runtime aspects are analogous to the variant above.

Containment Operation Reference Annotation Parameter to Marker Interface for x..1 References

This variant is for references with a cardinality of *0..1* or *1..1* and targets translated with the Marker Interface mechanism. In contrast to the variant for Type Annotation or Static Interface as targets, it leverages the marker interface to introduce type safety.

Example An example of the Containment Operation Reference Annotation Parameter to Marker Interface for x..1 References is shown in Figure 5.40. The meta model comprises a source class *Transition*, which owns a reference *target* to the class *State*. The target class specifies an attribute *name*, a *String*. The model defines an object of the class *Transition* with the name *scanItem*, which targets an object of the class *State* with the name *Scanning* with the reference *target*. This means that when this transition is executed, the next state is the *Scanning* state.

The code shows the Containment Operation Annotation that represents the containment reference towards the class *Transition*. This mechanism extends the annotation with an annotation parameter named after the reference name. The parameter's type is the marker interface that represents the target class.

For the model notation, the code shows a containment operation named `scanItem`. The annotation parameter declared in the meta model notation is set to the type `Scanning`, which represents the corresponding state with the Marker Interface mechanism.

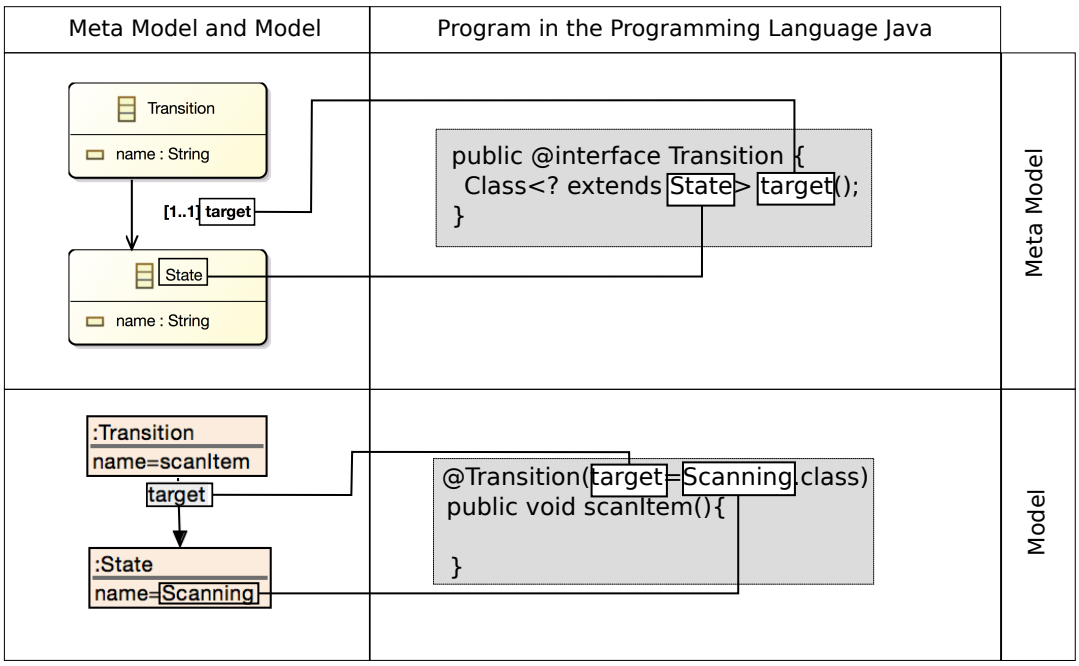


Figure 5.40: Example code for the mechanism Containment Operation Reference Annotation Parameter to Marker Interface for x..1 References

Formalization In the following, only the deviation of this variant's formalization from Definitions 56 and 57 is given.

Definition 60: Containment Operation Reference Annotation Parameter to Marker Interface for x..1 References - Meta Model Notation

The meta model notation of the Containment Operation Reference Annotation Parameter mechanism $P \xleftrightarrow[ContainmentOperationReferenceAnnotationParameter_{MI}^{x..1}]{represents} M_{Meta}$ deviates from the definition of $\xleftrightarrow[ContainmentOperationReferenceAnnotationParameter_{TA/SI}^{x..1}]{represents}$ as follows: follows:

Instead of representing the target class with the Type Annotation or the Static Interface mechanism, program structures P_{lib}^{target} exist, that represent the meta model M_{Meta}^{target} with the Marker Interface meta model notation. They represent a class target class with a marker interface. The program references these program structures.

$$P_{lib}^{target} \xleftrightarrow[MarkerInterface]{represents} M_{Meta}^{target}, class_{target} \in Classes_{M_{Meta}^{target}}, interface_{target} \in O_{P_{lib}^{target}},$$
$$interfaces_{target} \xrightarrow{instanceOf} \mathcal{I}, P_{lib}^{target} \in R_P$$

Instead of targeting any type or interface, the annotation parameter's type is the marker interface. *1..**.

$$ap.type \xrightarrow{references} interface_{target}$$

Definition 61: Containment Operation Reference Annotation Parameter to Marker Interface for x..1 References - Model Notation

The model notation of the Containment Operation Reference Annotation Parameter mechanism $P \xleftarrow[ContainmentOperationReferenceAnnotationParameter_{MI}^{x..1}]{represents} M_{Meta}$ deviates from the definition of $\xleftarrow[ContainmentOperationReferenceAnnotationParameter_{TA/SI}^{x..1}]{represents}$ as follows:

Instead of targeting objects translated with the Type Annotation mechanism or the Static Interface mechanism, program structures P_{lib}^{target} exist, that represent the model M^{target} with the Marker Interface mechanism, which represents a target object with a type. The program references these program structures.

$$P_{lib}^{target} \xleftarrow[MarkerInterface]{represents} M^{target}, o_{target} \in O_{M^{target}}, o_{target} \xrightarrow{instanceOf} class_{target},$$
$$e_{target} \in O_{P_{lib}^{target}}, e_{target} \xrightarrow{instanceOf} \mathcal{T}, P_{lib}^{target} \in R_P$$

Discussion For this variant to be applicable, the source object must be represented with the Containment Operation mechanism, the target class must be represented with the Marker Interface mechanism and the reference's cardinality must be *0..1* or *1..1*.

In contrast to the variant for targets translated with the Type Annotation or Static Interface mechanism, this variant creates type safety for the values of the annotation parameter using the marker interfaces. When the reference has a cardinality of *0..1*, its reference might have no target assigned to it. In that case, the same solution can be applied as described in the variant of this mechanism for the Type Annotation or Static Interface mechanism. The runtime aspects are analogous to the variants above. Other design and runtime aspects are analogous to in the said variant. The concept of this mechanism has been used by Balz, e.g. for target of transitions in state machines [Bal11, Section 4.1.2.2].

Containment Operation Reference Annotation Parameter to Marker Interface for x..* References

This variant is for references with a cardinality of *0..** or *1..** and targets represented with the Marker Interface mechanism. In contrast to the variant for *0..1* or *1..1* cardinalities, this variant uses an annotation parameter with an array type to represent multiple targets.

Example An example of the Containment Operation Reference Annotation Parameter to Marker Interface for x..* References is shown in Figure 5.41. The meta model defines a source class *Transition*, which owns a reference *contracts* to the class *Contract*. The target class specifies an attribute *name*, a *String* and is translated using the Marker Interface mechanism.

The model defines an object of the class *Transition* with the name *Ready*, which targets one object with the name *StartSaleContract*, and one object with the name *ScanItemContract* of the class *Contract* with the reference *contracts*. The reference declares contracts with pre and post conditions for the transition. Only when the preconditions of all contracts are fulfilled, the transition is executed. After execution, the transition guarantees that the post conditions of all contracts are true. The contracts are reusable. E.g. the *ScanItemContract* is used by this transition as well as the transition with the same name from the state *Scanning* to itself.

The code shows the Containment Operation Annotation that represents the containment reference towards the class *Transition*. This mechanism extends the annotation with an annotation parameter named after the reference name. The parameter references an array of the target marker interface.

For the model notation, the code shows a containment operation named `scanItem`. The annotation parameter declared in the meta model notation is set to an array that contains the types `StartSaleContract` and `ScanItemContract`, which represent the corresponding roles with the Marker Interface mechanism.

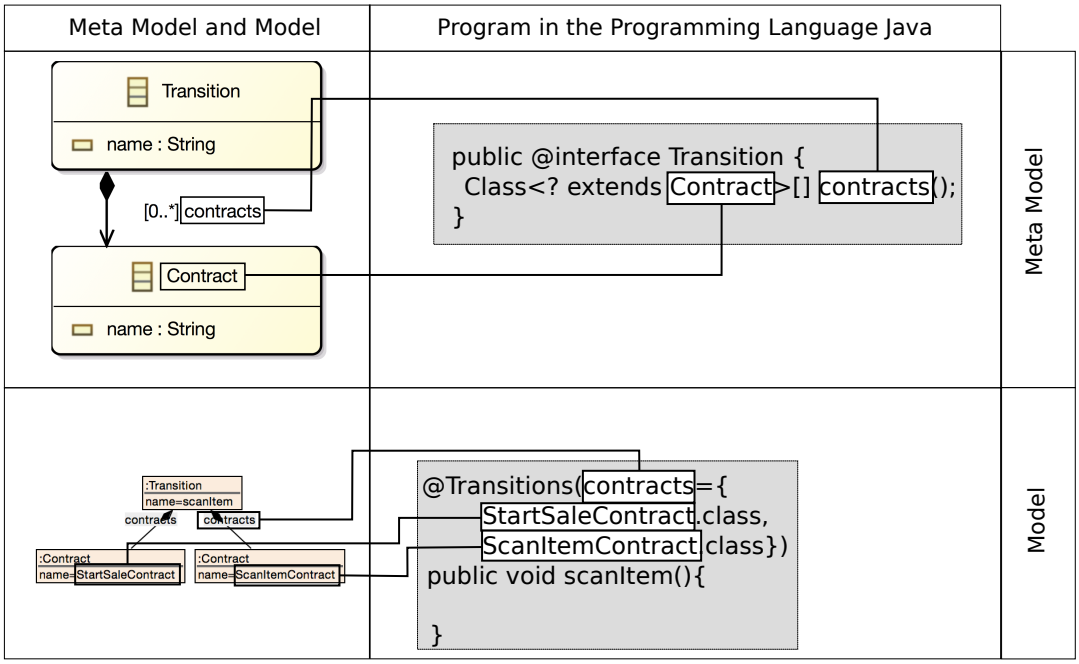


Figure 5.41: Example code for the mechanism Containment Operation Reference Annotation Parameter to Marker Interface for x..* References

Formalization In the following, only the deviation of this variant's formalization from Definitions 60 and 61 are given.

Definition 62: Containment Operation Reference Annotation Parameter to Marker Interface for x..* References - Meta Model Notation

The meta model notation of the Containment Operation Reference Annotation Parameter mechanism $P \xleftrightarrow[ContainmentOperationReferenceAnnotationParameter^{x..*}_{MI}]{represents} M_{Meta}$ deviates from the definition of $\xleftrightarrow[ContainmentOperationReferenceAnnotationParameter^{x..1}_{MI}]{represents}$ as follows:

Instead of a cardinality of either *0..1* or *1..1*, the reference has a cardinality of either *0..** or *1..**.

$$reference \xrightarrow{cardinality} 0..* \oplus reference \xrightarrow{cardinality} 1..*$$

The annotation parameter declared in the program is of an array type.

$$ap.isArrayType \xrightarrow{hasValue} true$$

Definition 63: Containment Operation Reference Annotation Parameter to Marker Interface for x..* References - Model Notation

The model notation of the Containment Operation Reference Annotation Parameter mechanism $P \xleftarrow[ContainmentOperationReferenceAnnotationParameter^{x..*}_{TA/SI}]{represents} M_{Meta}$ deviates from the definition of $\xleftarrow[ContainmentOperationReferenceAnnotationParameter^{x..1}_{TA/SI}]{represents}$ as follows:

Instead of the meta model notation for *0..1* or *1..1* references, this variant uses the meta model notation for *0..** or *1..** references.

$$P^{Reference}_{lib} \xleftrightarrow[ContainmentOperationReferenceAnnotationParameter^{x..*}_{MI}]{represents} M^{Reference}_{Meta}$$

Discussion For this variant to be applicable, the source object must be represented with the Containment Operation mechanism, the target class must be represented with the Marker Interface mechanism, and the reference's cardinality must be *0..** or *1..**.

As in the variant for *0..1* or *1..1* cardinalities, the target is defined by the annotation parameter value at design time. The assignment of targets within the code is also type safe. As in the variant for targets translated with the Type Annotation or Static Interface mechanism, when the reference has a cardinality of *0..**, it might have no target assigned. In that case, an empty array can be given as parameter value. Other design and runtime aspects are analogous to in the variants above.

Containment Operation Reference Parameter

The Containment Operation Reference Parameter mechanism requires the source class to be represented with a Containment Operation. The target class is represented with a type or

interface. An operation parameter, which represents the reference, is added to the containment operation.

Example An example of this mechanism is shown in Figure 5.42. The meta model comprises a source class *Transition*, which owns a reference *interfaces* to the class *Interface*. The target class specifies an attribute *name*, a *String*. The model defines an object of the class *Transition* with the name *scanItem*, which references one object of the class *Interface* with the name *IPrinter* and one with the name *IStoreServer*. These interfaces are provided or required by the component that is described using the state machine. The transition can invoke the executional semantics described by operations of these interfaces.

The code specifies the operation `scanItem`, which represents the source transition with the Containment Operation for Types mechanism. Its has parameters of the types `IPrinter` and `IStoreServer`, which represent the target objects with the Static Interface mechanism.

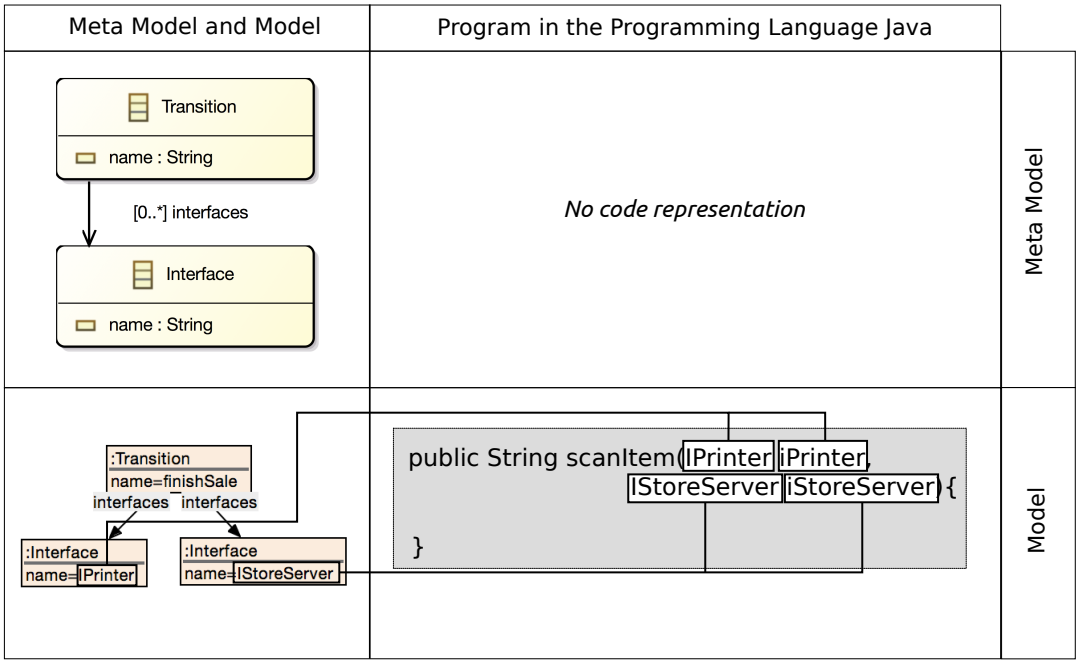


Figure 5.42: Example code for the Containment Operation Reference Parameter mechanism

Formalization The Containment Operation Reference Parameter mechanism is formalized as follows:

Definition 64: Containment Operation Reference Parameter - Meta Model Notation

The meta model notation of the Containment Operation Reference Parameter mechanism $P \xleftrightarrow[ContainmentOperationReferenceParameter]{represents} M_{Meta}$ is defined as follows:

Code structures exist that represent a source class with a Containment Operation meta model notation. The program references these code structures.

$$P_{lib}^{source} \xleftrightarrow[ContainmentOperationforTypes]{represents} M_{Meta}^{source}$$
$$\oplus P_{lib}^{source} \xleftrightarrow[ContainmentOperationforInterfaces]{represents} M_{Meta}^{source},$$
$$class_{source} \in Classes_{M^{source}}, P_{lib}^{source} \in R_P$$

Code structures exist that represent a target class with the Type Annotation or the Static Interface meta model notation.

$$P_{lib}^{target} \xleftrightarrow{represents} M_{Meta}^{target}$$
$$class_{target} \in Classes_{M_{Meta}^{target}}, element_{target} \in O_{P_{lib}^{source}},$$
$$element_{target} \xrightarrow{instanceOf} \mathcal{T} \oplus element_{target} \xrightarrow{instanceOf} \mathcal{I},$$
$$P_{lib}^{source} \in R_P$$

The meta model M_{Meta} comprises a reference between the source and the target class.

$$reference \in References_{M_{Meta}},$$
$$class_{source}.reference \xrightarrow{isOfType} class_{target}$$

Definition 65: Containment Operation Reference Parameter - Model Notation

For a program P and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Definition 64, the model notation of the Containment Operation Reference Parameter mechanism $(P, E) \xleftrightarrow[ContainmentOperationReferenceParameter]{represents} M$ is defined as follows:

Program structures P_{lib}^{source} exist, that represent the model M^{source} with a Containment Operation model notation. The program references these program structures.

$$(P_{lib}^{source}, E^{source}) \xleftrightarrow[ContainmentOperationforTypes]{represents} M^{source}$$
$$\oplus (P_{lib}^{source}, E^{source}) \xleftrightarrow[ContainmentOperationforInterfaces]{represents} M^{source},$$
$$object_{source} \in O_{M^{source}}, os_{source} \in O_{P_{lib}^{source}}, os_{source} \xrightarrow{instanceOf} \mathcal{OS}, P_{lib}^{source} \in R_P$$

Program structures P_{lib}^{target} exist, that represent the model M^{target} with the a model notation, that represents the target object with a type or interface. The target object has an identifying attribute. The program references these program structures.

$$(P_{lib}^{target}, E^{target}) \xleftarrow{represents} M^{target}, object_{target} \in O_{M_{target}},$$
$$element_{target} \in O_{P_{lib}^{target}}, object_{target} \xrightarrow{instanceOf} class_{target}, class_{target}.id \xrightarrow{isOfType} String,$$
$$element_{target} \xrightarrow{instanceOf} \mathcal{T} \oplus element_{target} \xrightarrow{instanceOf} \mathcal{I}, P_{lib}^{target} \in R_P$$

Program structures $P_{lib}^{Reference}$ exist, that represents the meta model $M_{Meta}^{Reference}$ with a the Containment Operation Reference Parameter meta model notation. The program references these program structures.

$$P_{lib}^{Reference} \underset{ContainmentOperationReferenceParameter}{\overset{represents}{\longleftrightarrow}} M_{Meta}^{Reference},$$
$$reference \in References_{M_{Meta}^{Reference}}, P_{lib}^{Reference} \in R_P$$

The source object references the target object with the reference.

$$(object_{source}, reference) \xrightarrow{references} object_{target}$$

The program declares an operation parameter for the target object, and adds it to the parameter list of the source operation signature. The parameter's type is the type or interface that represents the target object. The parameter's name equals the value of the identifying attribute of the target object.

$$op \in O_P, op \xrightarrow{instanceOf} \mathcal{OP}, op \xrightarrow{isOfType} element_{target},$$
$$os_{source}.\text{parameters} \xrightarrow{has} op, name(op) \qquad alue(object_{target}.id)$$

The model notation does not define an entry point.

$$E = \emptyset$$

Discussion For the Containment Operation Reference Parameter mechanism to be applicable, the source object must be represented with a Containment Operation mechanism, and the target class must be represented with a type or an interface.

At design time, the target object can be identified by the operation parameter type. When a source object references multiple objects with the same name, or same object multiple times with different references, only one of those references may be translated using this notation. Otherwise two parameters with the same name would be added to the parameter list. As this cannot be ensured when integration mechanisms are mapped to meta model elements, it should be avoided to use this mechanism for multiple references with the same source class.

At run time, the operation and its parameters are available to an execution runtime as well as to arbitrary program code. The execution semantics of the targeted object are available to the code in the containment operation's body. No other mechanism in this thesis provides this access when the source class is represented with the Containment Operation mechanism.

5.6.5 Attribute Representation

Integration mechanisms for attribute representations can be used to represent attributes in a meta model and attribute value assignments in a model with program code structures. Table 5.5 shows integration mechanisms for representing attributes in models with program code structures, and their requirements regarding the notations of the attribute owner. The first three integration mechanisms for attributes, that are defined in the following sections, are only applicable to attributes which are owned by classes, whose objects are represented with a type, e.g. with the mechanisms *Marker Interface* or *Type Annotation*. The last mechanism is applicable when containment operations are used as mechanism for the source class.

Attribute Representations	
Name	**Owner**
Constant Member Attribute	Represented as type
Attribute Annotation	Represented as type
Attribute Annotation Parameter	Represented as type
Containment Operation Attribute Annotation Parameter	Represented as containment operation

Table 5.5: An overview of integration mechanisms for representing attributes with program code structures

Constant Member Attribute

Attributes and attribute value assignment to objects can be represented using member attributes, when the corresponding class is represented by a type declaration, e.g. via the Marker Interface or the Type Annotation mechanisms. The member attribute is marked with an annotation to distinguish it from member attributes that do not represent model attributes.

Example An example of the Constant Member Attribute mechanism is shown in Figure 5.43. The meta model comprises the source class *ComponentType*, which owns an attribute *parallel* typed *Boolean*. The model defines an object of the class *ComponentType* with the name *StoreServer*. Its value for the attribute is *true*.

The code specifies the annotation `Attribute` for the meta model. The type `StoreServer` represents the source object with the Type Annotation mechanism. It owns a member attribute named `parallel`, typed `Boolean`. The annotation `Attribute` is attached to the member attribute for marking it a representation of a model attribute using this mechanism. The example uses additional modifiers from the Java language to make the attribute read-only and static.

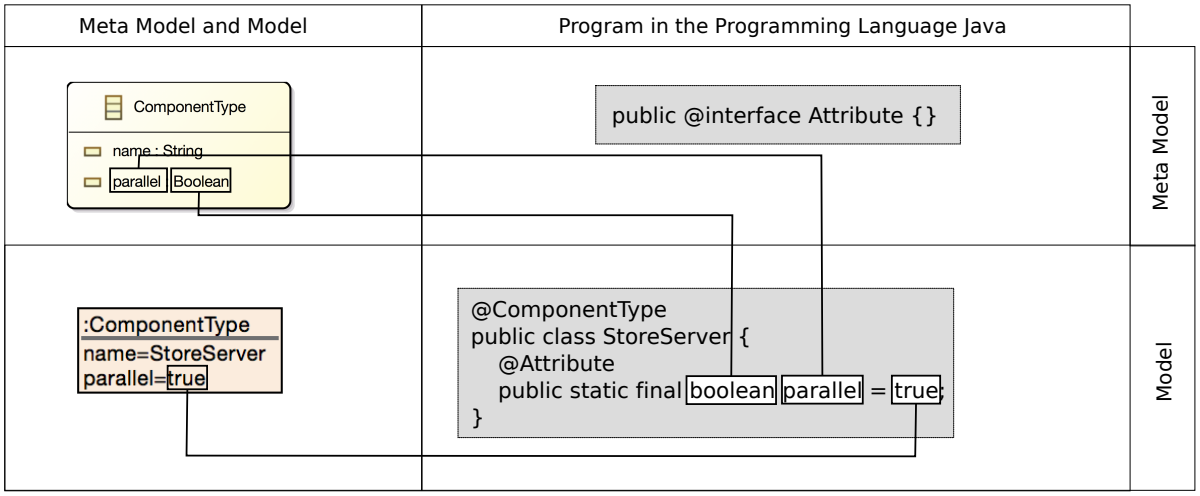


Figure 5.43: Example code for the Constant Member Attribute Mechanism

Formalization The Constant Member Attribute mechanism is formalized as follows:

Definition 66: Constant Member Attribute - Meta Model Notation

The meta model notation of the Constant Member Attribute mechanism $P \xleftarrow[ConstantMemberAttribute]{represents} M_{Meta}$ is defined as follows:
The meta model M_{Meta} defines an attribute.

$$attribute \in Attributes_{M_{Meta}}$$

The program declares an annotation.

$$annotation \in O_P, annotation \xrightarrow{instanceOf} \mathcal{A}$$

The annotation is named *Attribute*.

$$annotation.name \xrightarrow{hasValue} \text{Attribute}.$$

Definition 67: Constant Member Attribute - Model Notation

For a program P and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Definition 66, the model notation of the Constant Member Attribute mechanism $(P, E) \xleftarrow[ConstantMemberAttribute]{represents} M$ is defined as follows:
Program structures $P_{lib}^{Attribute}$ exist, that represent the meta model M_{Meta}^{Class} with the Constant Member Attribute meta model notation. The attribute is notated therein with

an annotation. The program references these program structures.

$$P_{lib}^{Attribute} \xleftrightarrow[ConstantMemberAttribute]{represents} M_{Meta}^{Attribute}, attribute \in Attributes_{M_{Meta}^{Attribute}},$$
$$annotation \in O_{P_{lib}^{Attribute}}, annotation \xrightarrow{instanceOf} \mathcal{A}, P_{lib}^{Attribute} \in R_P$$

Program structures P_{lib}^{Class} exist, that represent the model M^{Class} with a model notation that represents an object with a type. The program P references these program structures.

$$P_{lib}^{Class} \xleftrightarrow{represents} M^{Class}, object \in O_{M^{Class}}, object \xrightarrow{instanceOf} class,$$
$$type \in O_{P_{lib}^{Class}}, type \xrightarrow{instanceOf} \mathcal{T}, P_{lib}^{Class} \in R_P$$

The *attribute* is owned by *class*.

$$class \xrightarrow{has} attribute$$

The program P declares a member attribute ma, that is owned by *type*.

$$ma \in O_P, ma \xrightarrow{instanceOf} \mathcal{MA}, type \xrightarrow{has} ma$$

The member attribute's name and type equals the name and type of *attribute*. The marker annotation is assigned to the member attribute.

$$ma.name \xrightarrow{hasValue} name(attribute), \qquad ma.type \xrightarrow{hasValue} type(attribute),$$
$$annotation \xrightarrow{attachedTo} ma$$

The value assigned to the member attribute equals the value assigned to the model attribute.

$$ma.value \xrightarrow{hasValue} value(object.attribute)$$

The model notation defines the member attribute as entry point. I.e. it can be extended with annotations.

$$E = \{ma\}$$

Discussion For the Constant Member Attribute mechanism to be applicable, the owning class must be represented with a type declaration. This can be accomplished by using the Marker Interface or Type Annotation mechanism. The value can be read by a runtime using introspection mechanisms or by arbitrary code.

The example in Figure 5.43 uses Java's modifiers to define the character of the member attribute. The member attribute is declared *public* for it to be easily accessible by an execution runtime or other code. It is declared *static*, which means that the value is assigned on the type level instead of the instance level. This mechanism represents the model attribute with the declaration of the (code) member attribute, not by value assignments to type instances. Therefore a static member attribute represents the attribute value assignment better.

Following the formalization, it is allowed to change the attribute value at runtime. This seems unexpected because the models considered in the Model Integration Concept are of a

static nature. When the underlying programming language has mechanisms for declaring read only member attributes, such as the *final* modifier from Java used in the example, these can be used to protect the value.

Attribute Annotation Parameter

The Attribute Annotation Parameter mechanism uses parameters of annotations to represent model attributes and their values. This requires that the class that owns the attribute is notated with the Type Annotation or with the Static Interface mechanism.

Example An example of the Attribute Annotation Parameter mechanism is shown in Figure 5.44. The meta model comprises the source class *ComponentType*, which owns an attribute *version* typed *String*. The model defines an object of the class *ComponentType* with the name *StoreServer*. Its value for the attribute is 1..1.

The source class is translated using the Type Annotation mechanism. I.e. it has an annotation attached that declares its model class. This mechanism extends this annotation with an annotation parameter named and typed after the attribute. The type *StoreServer* sets the attribute value in the attached annotation parameter.

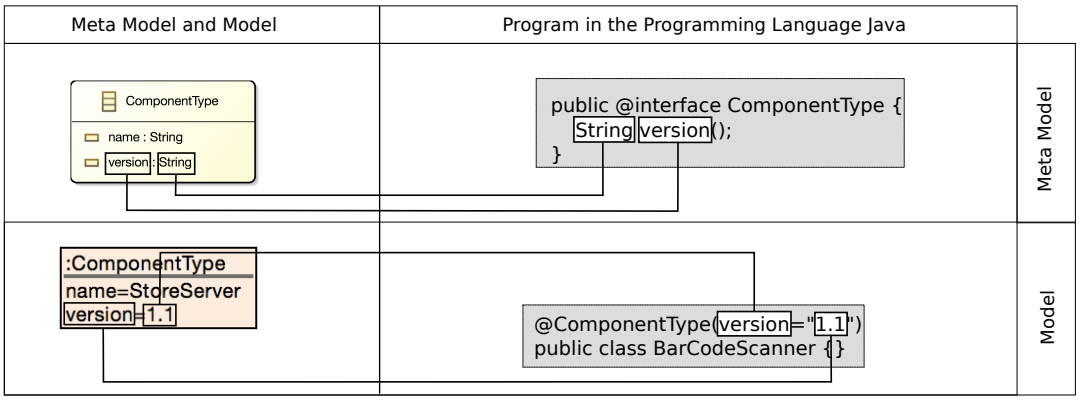


Figure 5.44: Example code for the Attribute Annotation Parameter Mechanism

Formalization The Attribute Annotation Parameter mechanism is formalized as follows:

Definition 68: Attribute Annotation Parameter - Meta Model Notation

The meta model notation of the Attribute Annotation Parameter mechanism $P \xleftrightarrow[AttributeAnnotationParameter]{represents} M_{Meta}$ is defined as follows:

Program structures P_{lib} exist, that represent the meta model M_{Meta}^{Class} with the Type Annotation meta model notation. The owning class is notated with an annotation by the

Type Annotation mechanism.

$$P_{lib} \xleftrightarrow[TypeAnnotation]{represents} M_{Meta}^{Class}, class \in Classes_{M_{Meta}^{Class}}, annotation \in O_{P_{lib}}, P_{lib} \in R_P$$

The meta model M_{Meta} comprises an attribute.

$$attribute \in Attributes_{M_{Meta}}$$

The *attribute* is owned by *class*.

$$class \xrightarrow{has} attribute$$

The program declares an annotation parameter, which is named and typed after the meta model's attribute name and type, and owned by the annotation.

$$ap \in O_P, ap \xrightarrow{instanceOf} \mathcal{AP}, \qquad ap.name \xrightarrow{hasValue} name(attribute),$$
$$ap.type \xrightarrow{hasValue} value(attribute.type), \qquad annotation.parameters \xrightarrow{references} ap$$

Definition 69: Attribute Annotation Parameter - Model Notation

For a program P and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Definition 68, the model notation of the Attribute Annotation Parameter mechanism $(P, E) \xleftrightarrow[AttributeAnnotationParameter]{represents} M$ is defined as follows:

Program structures $P_{lib}^{Attribute}$ exist, that represent the meta model $M^{Attribute}$ with the meta model notation $\xleftrightarrow[AttributeAnnotationParameter]{represents}$. An attribute is notated therein with an annotation parameter. The program references these program structures.

$$P_{lib}^{Attribute} \xleftrightarrow[AttributeAnnotationParameter]{represents} M_{Meta}^{Attribute},$$
$$attribute \in Attributes_{M_{Meta}^{Attribute}}, ap \in O_{P_{lib}^{Attribute}}, ap \xrightarrow{instanceOf} \mathcal{AP},$$
$$P_{lib}^{Attribute} \in R_P$$

Program structures P_{lib}^{Class} exist, that represent the model M^{Class} with the Type Annotation model notation. An object is notated therein with an annotation attached to a type. The program references these program structures.

$$(P_{lib}^{Class}, E) \xleftrightarrow[TypeAnnotation]{represents} M^{Class}; object \in O_{M^{Class}}; object \xrightarrow{instanceOf} class;$$
$$type, annotation \in O_{P_{lib}^{Class}}; type \xrightarrow{instanceOf} \mathcal{T}; annotation \xrightarrow{instanceOf} \mathcal{A};$$
$$P_{lib}^{Attribute}, P_{lib}^{Class} \in R_P$$

The value assigned to the member attribute equals the value assigned to the annotation

parameter for the object.

$$object.attribute \xrightarrow{hasValue} value(type.annotation.ap)$$

The model notation does not define an entry point.

$$E = \emptyset$$

Discussion For the Attribute Annotation Parameter mechanism to be applicable, the containing class must be represented by the Type Annotation mechanism. The value can be read by a runtime using introspection mechanisms.

Attribute Annotation

The Attribute Annotation mechanism is closely related to the Attribute Annotation Parameter mechanism. Instead of adding an annotation parameter to an existing annotation, it uses an own annotation for an attribute. Also, the underlying notation for the class can be any notation that translates a class to a type or interface.

Example The meta model defines the source class *State*, which owns an attribute *immediate* typed *Boolean*. The model defines an object of the class *State* with the name *Scanning*. Its value for the attribute is *true*. This means that the state machine will immediately fire the next transition, when it reaches this state.

The attribute is represented in the code with an annotation that is named after the attribute. The annotation has a default parameter typed after the attribute. The annotation is attached to the type which represents the source class. Its attribute value is set as value of the default parameter.

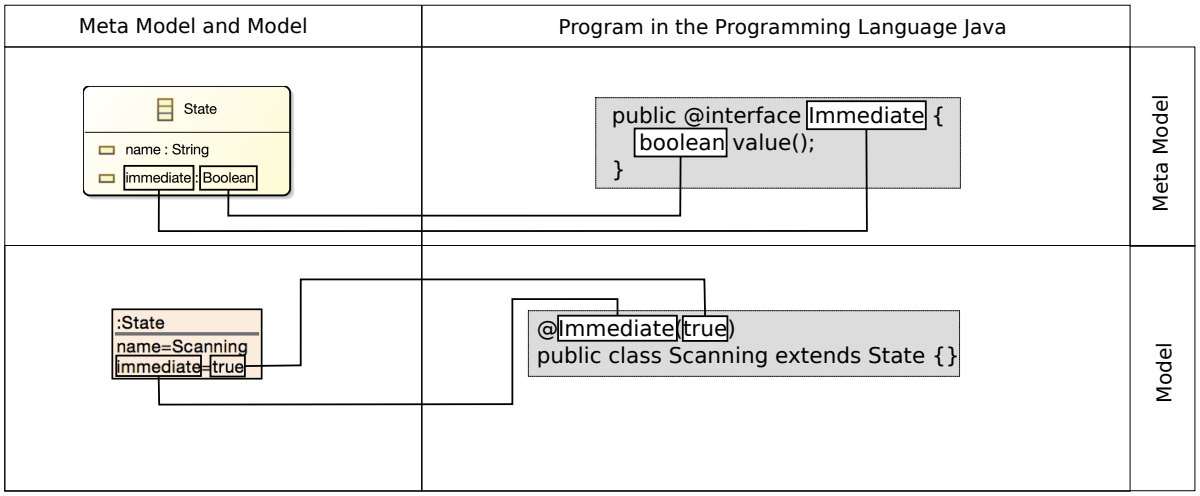


Figure 5.45: Example code for the Attribute Annotation Mechanism

Formalization The Attribute Annotation mechanism is formalized as follows:

Definition 70: Attribute Annotation - Meta Model Notation

The meta model notation of the Attribute Annotation mechanism $P \xleftrightarrow[AttributeAnnotation]{represents} M_{Meta}$ is defined as follows:

The meta model M_{Meta} defines an attribute.

$$attribute \in Attributes_{M_{Meta}}$$

The program declares an annotation and an annotation parameter.

$$a, ap \in O_P, a \xrightarrow{instanceOf} \mathcal{A}, ap \xrightarrow{instanceOf} \mathcal{AP}$$

The annotation is named after the meta model's attribute name. The annotation parameter is the default parameter. Its type equals the model attribute's type.

$$\begin{aligned} a.name &\xrightarrow{hasValue} name(attribute), \\ a.parameters &\xrightarrow{references} ap, \\ a.defaultParameter &\xrightarrow{references} ap, \\ ap.type &\xrightarrow{hasValue} type(attribute) \end{aligned}$$

Definition 71: Attribute Annotation - Model Notation

For a program P and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Definition 70, the model notation of the Attribute Annotation mechanism $(P, E) \xleftarrow[AttributeAnnotation]{represents} M$ is defined as follows:

Program structures $P_{lib}^{Attribute}$ exist, that represent the meta model $M_{Meta}^{Attribute}$ with the meta model notation $\xleftrightarrow[AttributeAnnotation]{represents}$. An attribute is notated therein with an annotation and its default parameter. The program references these program structures.

$$\begin{aligned} P_{lib}^{Attribute} &\xleftrightarrow[AttributeAnnotation]{represents} M_{Meta}^{Attribute}; \\ attribute \in Attributes_{M_{Meta}^{Attribute}}; a, ap \in O_{P_{lib}^{Attribute}}; \\ a \xrightarrow{instanceOf} \mathcal{A}; ap \xrightarrow{instanceOf} \mathcal{AP}; P_{lib}^{Attribute} \in R_P \end{aligned}$$

Program structures P_{lib}^{Class} exist, that represent the model M^{Class} with a model notation that represents an object with a type. The program references these program structures.

$$\begin{aligned} (P_{lib}^{Class}, E^{Class}) \xleftarrow{represents} M^{Class}, object \in O_{M^{Class}}, object \xrightarrow{instanceOf} class, \\ class \xrightarrow{has} attribute, type \in O_{P_{lib}^{Class}}, type \xrightarrow{instanceOf} \mathcal{T}, P_{lib}^{Class} \in R_P \end{aligned}$$

The program attaches the annotation to the type. The value assigned to the attribute equals the value assigned to the default parameter.

$$object.attribute \xrightarrow{hasValue} value(type.a.ap)$$

The model notation does not define an entry point.

$$E = \emptyset$$

Discussion For the Attribute Annotation mechanism to be applicable, the containing class must be represented by a type declaration. This can be accomplished by using the Marker Interface or Type Annotation mechanism. The value can be read by a runtime using introspection mechanisms.

Containment Operation Attribute Annotation Parameter

The mechanisms for attributes above require on the target being translated as a type or interface. When a class is represented with an operation, e.g. using the Containment Operation mechanism, these mechanisms are not applicable. The Containment Operation Attribute Annotation Parameter mechanism allows for specifying targets for references of these classes, be leveraging the annotation introduced by the Containment Operation mechanism. This mechanism is the equivalent of the Containment Operation Reference Annotation Parameter mechanism for attributes.

Example An example of this mechanism is shown in Figure 5.46. The meta model defines a source class *Operation*, which has an attribute *timeResourceDemand*, a *String*. The model defines an object of the class *Operation* with the name *scanItem*, and a value for the said attribute. The value is a String-based definition of a probability mass function for the time necessary until an invocation of this operation is finished as it is used in the PCM [BKR09]. In this example with a probability of 0.1 the operation requires 10 milliseconds, with a probability of 0.5 it requires 20 milliseconds, and with a probability of 0.2 it requires 50 milliseconds.

The code shows the Containment Operation Annotation that represents the containment reference towards the class *Operation*. This mechanism extends the annotation with an annotation parameter. The parameter is named and typed after the attribute.

The operation *scan* represents the object that is translated with the Containment Operation mechanism. The value of the attribute is assigned to the annotation parameter of the containment operation annotation.

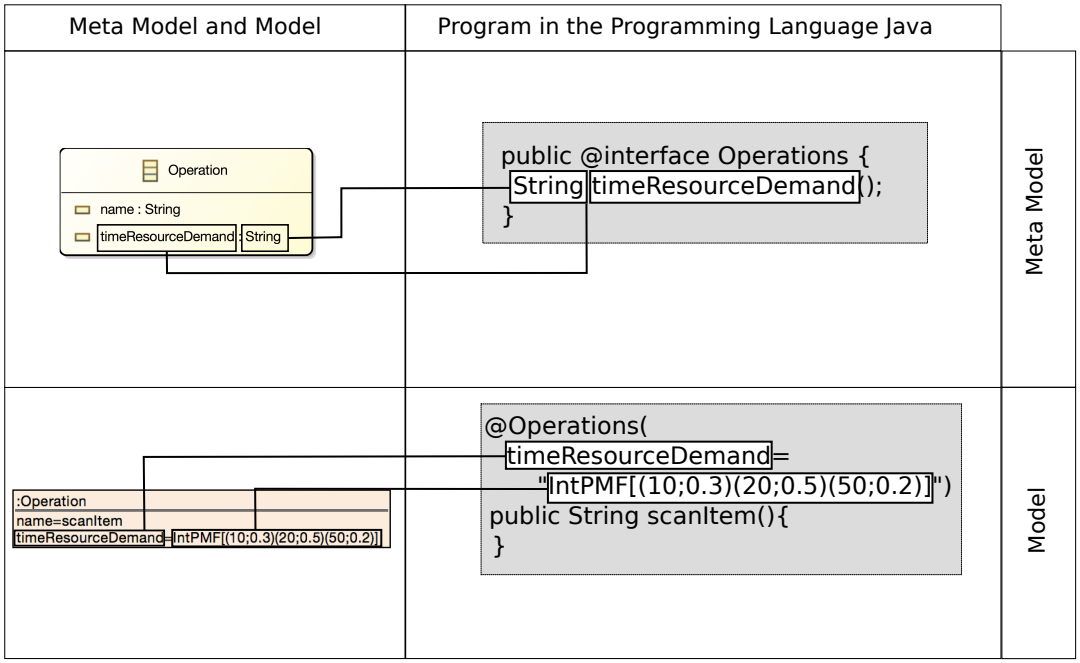


Figure 5.46: Example code for the mechanism Containment Operation Attribute Annotation Parameter

Formalization The Containment Operation Attribute Annotation Parameter mechanism is formalized as follows:

Definition 72: Containment Operation Attribute Annotation Parameter - Meta Model Notation

The meta model notation of the Containment Operation Attribute Annotation Parameter mechanism $P \xleftrightarrow[ContainmentOperationAttributeAnnotationParameter]{represents} M_{Meta}$ is defined as follows:

Program structures P_{lib}^{owner} exist, that represent the meta model M_{Meta}^{owner} with the Containment Operation meta model notation, that represents a reference and a target class with an annotation. The program P references these program structures.

$$P_{lib}^{Class} \xleftrightarrow[ContainmentOperationforTypes]{represents} M_{Meta}^{Class}$$

$$\oplus P_{lib}^{Class} \xleftrightarrow[ContainmentOperationforInterfaces]{represents} M_{Meta}^{Class},$$

$$reference_{Class} \in References_{M_{Meta}^{Class}}, class \in Classes_{M_{Meta}^{Class}},$$

$$annotation \in A_{P_{lib}^{Class}}, P_{lib}^{Class} \in R_P$$

The meta model M_{Meta} defines an attribute that is owned by the class.

$$attribute \in Attributes_{M_{Meta}}, class_{source} \xrightarrow{has} attribute$$

The program declares an annotation parameter. It is owned by the annotation, and named and typed after the model attribute.

$$\begin{aligned} ap \in O_P, ap &\xrightarrow{instanceOf} \mathcal{AP}, \\ annotation.\text{parameters} &\xrightarrow{references} ap, \\ ap.name &\xrightarrow{hasValue} name(attribute), \\ ap.type &\xrightarrow{hasValue} type(attribute) \end{aligned}$$

Definition 73: Containment Operation Attribute Annotation Parameter - Model Notation

For a program P and a model $M \xrightarrow{instanceOf} M_{Meta}$, with M_{Meta} defined as in Definition 72, the model notation $(P, E) \xleftarrow[ContainmentOperationAttributeAnnotationParameter]{represents} M$ is defined as follows:

Program structures P_{lib}^{source} exist, that represent the model M^{source} with the Containment Operation meta model notation, which represent a source object with an operation signature. The program references these program structures.

$$(P_{lib}^{source}, E^{source}) \xleftarrow[ContainmentOperationforTypes]{represents} M^{source}$$

$$\oplus(P_{lib}^{source}, E^{source}) \xleftarrow[ContainmentOperationforInterfaces]{represents} M^{source},$$

$$o_{source} \in O_{M^{source}}, os \in O_{P_{lib}^{source}}, os \xrightarrow{instanceOf} \mathcal{OS}, P_{lib}^{source} \in R_P$$

Program structures $P_{lib}^{Attribute}$ exist, that represent the meta model $M_{Meta}^{Attribute}$ with the meta model notation $\xleftrightarrow[ContainmentOperationAttributeAnnotationParameter]{represents}$. An attribute is notated therein with an annotation parameter. The corresponding annotation is attached to the operation signature. The program references on these program structures.

$$\begin{aligned} P_{lib}^{Attribute} &\xleftrightarrow[ContainmentOperationAttributeAnnotationParameter]{represents} M_{Meta}^{Attribute}, \\ attribute \in Attributes_{M_{Meta}^{Attribute}}, annotation, ap &\in O_{P_{lib}^{Attribute}}, \\ annotation &\xrightarrow{instanceOf} \mathcal{A}, ap \xrightarrow{instanceOf} \mathcal{AP}, \\ annotation.\text{parameters} &\xrightarrow{references} ap, annotation \xrightarrow{attachedTo} os, P_{lib}^{Attribute} \in R_P \end{aligned}$$

The attribute's value for the source object equals the value assigned to the parameter in

the attached annotation.

$$os.annotation.ap \xrightarrow{hasValue} value(o_{source}, attribute)$$

The model notation defines no entry point

$$E = \emptyset$$

Discussion For the Containment Operation Attribute Annotation Parameter mechanism to be applicable, the source object must be represented with the Containment Operation mechanism. In the mechanisms presented in this thesis, this is the only way to declare attributes of objects that are translated as Containment Operations. During design time, the value is defined by the annotation parameter value.

An attribute might have no value assigned. In that case, no value would be assigned to the parameter of the attached annotation, which contradicts Definition 31. In that case a value has to be declared that represents the empty target, e.g. an empty string, 0, or false, depending on the attribute's type. Java allows for setting default values for annotation parameters. This concept could be used to set such a value as default value, which is known to a translation or execution runtime. This might be unwanted, because an the semantics of an empty string and a null value for example might differ in the application.

Analogously to the Containment Operation Reference Annotation Parameter mechanism, an execution runtime can access the value using introspection mechanisms, and use it e.g. to evaluate whether the Containment Operation should be invoked, or for evaluating these values during a performance monitoring. The program code within the operation can access the value using the same introspection mechanisms, but the value is not available directly, e.g. via an operation parameter.

It would also be possible to add a parameter to the operation, which is typed and named after the attribute. An execution runtime could then inject the respective value into its calls. However, the operation and its parameters are available for invocation for arbitrary code, which could set arbitrary parameter values. Therefore the parameter value at runtime could differ from the value at design time, which is not intended here.

5.6.6 Summary

This section described and defined a set of integration mechanisms in the boundaries of the language definition presented in Section 5.4.4. These integration mechanisms can be used as templates for creating notations for specific languages. The existence of defined mechanisms allows for reasoning about integrations, and for evaluating the advantages and disadvantages of the use of particular integration mechanisms within the models to be mapped.

The presented mechanisms have specific advantages and disadvantages as described in the individual sections. The selection of integration mechanisms depends on how the resulting code elements should be usable by other code and by execution runtimes. A provided component interface should be represented using the Static Interface mechanism (the interface) and the Static Interface Implementation mechanism (the provision) to be usable as expected in the code. If the provision is e.g. represented with the Containment Operation mechanism, operations in the interface could only be represented as types targeted in an operation parameter or annotation, which would not meet developer expectations.

The presented list of integration mechanisms is not complete. New mechanisms can be added, which probably have other specific advantages or disadvantages. The programming language definition 15 is based on the Java Language Specification [GJS+15]. Most mechanisms require the concept of annotations, which limits the applicability to other languages. For defining integrations that are better usable with other languages, it might be necessary to revisit the underlying programming language definition, for creating a foundation that is better usable with the respective programming language.

5.7 Development of Model-to-Code/Code-to-Model Transformations and Execution Runtimes

For the integration of meta models and models with program code using the notations described above, it is required to have a meta model of the architecture implementation language. This is usually not the case. To use a specific architecture implementation language with the Model Integration Concept, the following steps need to be executed. Figure 5.47 accompanies the description as an overview. The figure shows the elements created in the steps described below. The arrows between the elements declare the data flow.

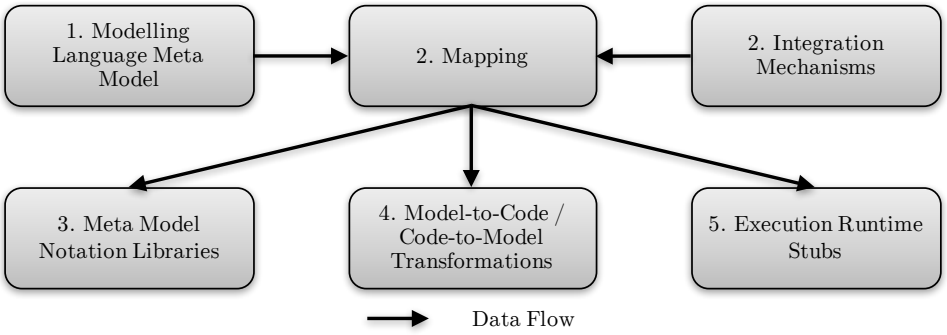


Figure 5.47: An overview of the elements, that need to be created during the development of code-to-model and model-to-code transformations and execution runtime stubs in the Model Integration Concept

1. A meta model that follows the Definitions 1 to 8 have to be created, based on the architecture implementation language's specification.

2. A mapping has to be created between the classes, attributes, and references in the meta model, and integration mechanisms. Where necessary, new integration mechanisms have to be created.

3. A code library must be implemented, that represents the meta model elements in the meta model notation of the integration mechanisms. This task can be automated, by implementing a generic tool that takes a meta model and the chosen integration mechanisms as parameter, and creates a corresponding code library. Such a tool has been developed in the context of this thesis (see Section 9.3.3).

4. Bidirectional model-code transformations must be implemented, that can translate existing code into a model representation based on the meta model and the chosen integration mechanisms, and that can translate changes in a model to the code. These transformations have the following functional requirements:
   a) For a given code element, a corresponding model element must be created.
   b) When two model elements are given, one from before a change and one from after a change, the following changes must be made to the program code:
      i. When the model element from before a change is empty, new program code must be created.
      ii. When the model element from after a change is empty, the corresponding program code must be deleted.
      iii. When both model elements are available and differ, the corresponding program code must be changed according to the integration mechanism.

   For formally defined integration mechanisms like those presented in Section 5.6, this task can be automated by implementing a generic tool that takes a meta model and the chosen integration mechanisms as a parameter, and creates a corresponding set of translations. For executing these translations an execution framework is necessary. Such tools and frameworks have been developed in the context of this thesis (see Sections 9.3.4 and 9.2).

5. An execution runtime must exist, that is capable of executing the execution semantics. Following Balz' analysis [Bal11, Section 3.1.4], an execution runtime can (a) interpret structures found in the integrated model. This allows for behaviour that is defined on the level of the modelling language. E.g. the general behaviour to instantiate component types can be defined this way, when the behaviour is independent from the specific component type. It must also (b) execute semantics defined within in entry points of the notations. E.g. when a component type is started, a specific operation of type declaration can be invoked. Each component type may declare an individual start-up operation.

   An execution framework can be either available separately from the integrated model, as it is defined by Balz [Bal11, Section 3.2.3], or be part of the integration mechanism's code structure. This is shown in the CoCoME component case study in Section 10.2.3.

For existing standardized architecture implementation languages used in practice, libraries often already exists as APIs. These APIs can be used to mark code elements e.g. as beans in JEE. In that case, the meta model notation is already fixed, which means that steps 2 and 3 are not needed. For these languages the execution runtime also typically already exits in form of a reference implementation and possibly further implementations of the specification. Therefore step 5 is also unnecessary in these cases.

It should be noted that architecture implementation languages use a variety of description styles for their model elements, including annotations, marker interfaces, but also e.g. XML configuration files. These mechanisms have to be analyzed and understood. It is possible to describe notations for other languages than programming languages. To do so, the language must be expressed using the formalism for meta models, or the formalism must be extended correspondingly. This thesis focuses on notations for object-oriented programming languages.

5.8 Summary

This chapter described the Model Integration Concept as a central part of the Explicitly Integrated Architecture approach and showed how it integrates model information with program code. First the foundational elements of the concept were defined: language meta models and models thereof, of both, modelling languages and programming languages. Then meta model notations and model notations were defined, which formally describe how a meta model or a model is represented unambiguously with program code structures. Integration mechanisms are templates for notations, and can be used in combination with a specific meta model to build specific notations. A process was described, how translations between meta models and program code structures of architecture implementation models can be developed using the Model Integration Concept. The role of the Model Integration Concept in the Explicitly Integrated Architecture approach is to create a bidirectional mapping between the program code and the architecture models that ar represented in an architecture implementation language or in the intermediate languages. The next chapter describes the intermediate language.

6 Intermediate Architecture Description Language

This chapter describes the Intermediate Architecture Description Language (IAL) as a translation model language between architecture implementation models and architecture specification models. Figure 6.1 highlights the role of the IAL within the proposed solution.

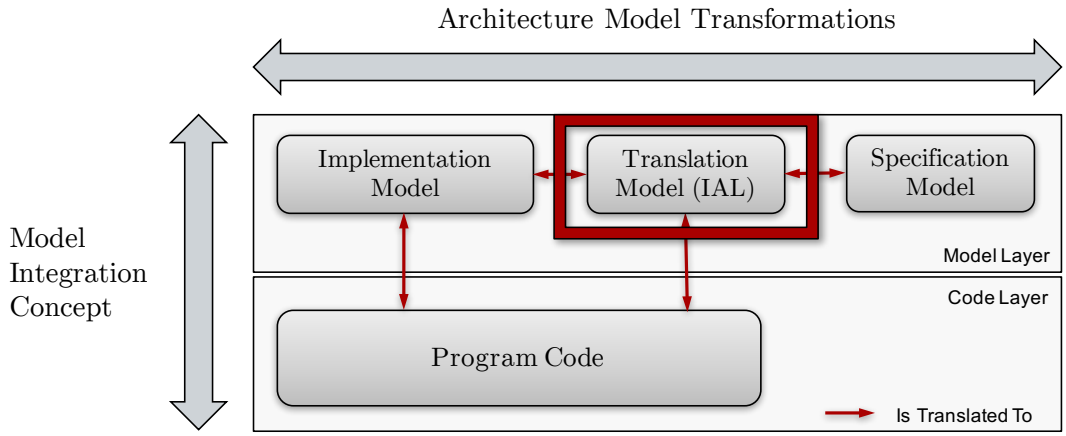


Figure 6.1: The Intermediate Architecture Description Language highlighted in the overview of the proposed solution

The IAL is used to represent architecture information independently from the architecture specification and implementation languages. It has the role to increase the interoperability of the Explicitly Integrated Architecture approach with different architecture languages, and to increase its evolvability. Section 6.1 states requirements towards the IAL to fulfill its role. A central requirement is the extensibility of the language. Section 6.2 describes concepts for building extensible languages and discusses why existing extensible languages are not suitable for the Explicitly Integrated Architecture Process. The then following sections describe the language developed for the process.

6.1 Requirements Towards a Translation Model Language

The IAL has two roles in the Explicitly Integrated Architecture Process. First, it assists language interoperability by relaxing the n:m relationship between architecture specification and implementation languages to a n:1:m relationship. It decouples the translations of the languages from each other within the approach, to make the aspect of architecture model transformations conceptually more sound. It decreases the number of transformation rules

necessary to be specified when a new language is integrated into the Explicitly Integrated Architecture Process. Second, it strengthens the evolvability of the approach. The IAL must be able to cover the concepts of any architecture languages, including future ones. The language can fulfill this requirement, when it is possible to extend the language's meta model with new elements, and to add properties to existing elements.

The IAL is required to model multiple concerns regarding its elements at the same time. E.g. a component type has to be declared stateful or stateless, part of a hierarchy or not, and subject to performance and security constraints, at the same time.

Architecture specification languages and architecture implementation languages have few commonalities [Mü10]. A minimal set of element types in architecture languages are usually components and interfaces. The interconnection between components is often an essential part of architecture descriptions. Component interconnections are usually enabled via interfaces that are provided and required by components. The actual connection is then modelled with a connector between two components with compatible interfaces. While some architecture languages do not explicitly model connectors (e.g. [MDEK95, vOvdLKM00]), these languages interconnect components via shared interfaces, which implies a connector, even if no first class element with such a name exists. Therefore, it must be possible to model components, interfaces, and connectors with the IAL.

At last, technical constraints due to the implementation of the process require the IAL to be easily usable with tools for Ecore meta models, for making the Ecore ecosystem available for implementing the IAL, and for parsing and creating models thereof. In summary, we state the following requirements towards an IAL for the Explicitly Integrated Architecture Process:

IL-R1 It must be possible to extend the meta model with new first class entities.

IL-R2 It must be possible to extend existing elements of the meta model with new properties.

IL-R3 It must be possible to model multiple concerns regarding a meta model element simultaneously.

IL-R4 It must be possible to represent components, interfaces, and connectors.

IL-R5 The meta model must be easily usable with tools working with Ecore meta models.

Requirement IL-R4 is fulfilled by most architecture languages. Requirement IL-R5 is a technical constraint. Where necessary, adapters could be developed to make languages usable with Ecore enabled tools. The requirements IL-R1, IL-R2, and IL-R3 are the most limiting factor for finding a suitable language. Section 6.2 describes the strategies and a set of languages inspected regarding these requirements, and explains why these languages are not suitable for the Explicitly Integrated Architecture Process.

6.2 Strategies and Language Concepts for Extensibility

The requirement for integrating new first class entities in meta models can only be fulfilled by extensible languages. Three strategies are identified and evaluated in the context of this thesis, how this requirement can be fulfilled. The strategies are described in the following sections.

6.2.1 Placeholders

Special abstract syntax elements can be used as placeholders for arbitrary information. Acme [GMW97] is a representative language for using placeholders. It has been intended to be an architecture interchange language. It is primarily used in automated processes as intermediate language, but has moved to be an Architecture Description Language (ADL) [TMD09, p. 230]. Acme has a set of basic elements for modelling architectures, primarily components, interfaces (called *ports* in Acme), and connectors. In architectures modelled with Acme, elements can be extended with arbitrary key-value pairs called *properties*.

Listing 6.1 gives an example for such a placeholder in Acme. The example declares a system `CoCoME` with two components. The component `cashBox` declares a property `JavaImplementation`, which states a class that implements the component.

```
System CoCoME = {                                                    1
  Component cashBox = {                                              2
    Port receiveProductCode;                                         3
    Property JavaImplementation: class = org.cocome.CashBox          4
  }                                                                  5
  Component barcodeScanner = {                                       6
    Port scanBarcode;                                                7
  }                                                                  8
  Connector scanEvent = {                                            9
    Role caller;                                                     10
    Role callee;                                                     11
  }                                                                  12
  Attachment cashBox.receiveProductCode to scanEvent.callee;         13
  Attachment barcodeScanner.scanBarcode to scanEvent.caller;         14
}                                                                    15
```

Listing 6.1: Example for placeholders with *Acme*

Acme, as a representative of architecture with extensions, does not fulfill all requirements stated in Section 6.1.

IL-R1 It must be possible to extend the meta model with new first class entities.

It is not possible to extend the language with new first class entities, because a property is always owned by another first class entity.

IL-R2 It must be possible to extend existing elements of the meta model with new properties.

It is possible to extend language elements with new properties with the *properties* mechanism. However, the concept implies that the extending properties have well-defined schema. The contents and its semantics are unclear.

IL-R3 It must be possible to model multiple concerns regarding a meta model element simultaneously.

The strategy allows for modelling multiple concerns simultaneously by adding arbitrary information within the placeholder.

IL-R4 It must be possible to represent components, interfaces, and connectors.

First class entities exist for components, interfaces (called ports), and connectors.

IL-R5 The meta model must be easily usable with tools working with Ecore meta models.

While it is possible to create an Ecore meta model for the Acme language, no such meta model already exists.

The placeholder strategy allows to extend elements with arbitrary information. It is flexible to use. It is however not part of the strategy to define a schema for the additional information within the boundaries of the underlying language. A supplemental technology, such as schemata, a parser etc. are necessary to use this strategy in an automated fashion. Due to these disadvantages this strategy is not chosen for the IAL.

6.2.2 Meta Model Extensions

Languages with meta model extensions declare extension points in meta models, where—on a model level—arbitrary extensions can be integrated. xADL [DvdHT02] is a representative language for using meta model extensions. The meta model of xADL is based on XML schemata. xADL defines a core XML schema, which is extended by schemata for specific architectural concerns. These extending schemata reference meta model elements from the core, or from other extending schemata. Depending on the requirements towards the architecture of a specific system, a set of chosen XML schemata is chosen to build the meta model of the architecture language. An architecture is designed using xADL by creating XML documents that are valid regarding the set of XML schemata. One of the predefined XML schemata in xADL is *structure-3.0.xsd*. It comprises language elements for modelling components, interfaces, and connectors. Other predefined schemata concern e.g. behavioural descriptions with state charts or variability. Other concerns can be developed and integrated as needed.

XML Schema allows for using abstract types and extensions of types, similar to abstract classes and subclasses in object-oriented programming languages. Subclassing uses inheritance to extend classes with functionality. With subclassing, classes can inherit attributes and references from parent classes. The children extend their parents with their own attributes and references. This allows for using subclasses (in terms of XML *subtypes*) for extending the classes in xADL languages. xADL uses this feature for its extension mechanism.

The listings 6.2 to 6.4 give an example using XML and the definition of extensions with subclasses. Listing 6.2 is an excerpt of xADL's *structure* schema. It declares a type *structure*, which contains components, connectors, and links. `Structures` (as well as many other types in xADL) contain elements of the abstract type `core:Extension` (see line 16 of Listing 6.2). Here new elements of other XML schemata can be added. The example in Listing 6.3 shows one such extension. The *implementation* schema provides a subtype of the `core:Extension` type, that may provide implementation details. The XML document in Listing 6.4 shows the declaration of a simple architecture with xADL. The component `CashBox` contains implementation details using the extension mechanism.

```
<xs:schema xmlns="http://www.archstudio.org/xadl3/schemas/structure-3.0.xsd"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:core="http://www.archstudio.org/xadl3/schemas/xadlcore-3.0.xsd"
           targetNamespace="http://www.archstudio.org/xadl3/schemas/structure-3.0.xsd"
           elementFormDefault="qualified"
           attributeFormDefault="qualified">

  <xs:import namespace="http://www.archstudio.org/xadl3/schemas/xadlcore-3.0.xsd"
             schemaLocation="https://raw.github.com/isr-uci-edu/ArchStudio5/master/org
                 archstudio.xadl3.xadlcore/model/xadlcore-3.0.xsd"/>

  <xs:complexType name="Structure">
    <xs:sequence>
      <xs:element name="component" type="Component" minOccurs="0" maxOccurs="unbounded"
    />
      <xs:element name="connector" type="Connector" minOccurs="0" maxOccurs="unbounded"
          />
      <xs:element name="link" type="Link" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="ext" type="core:Extension" minOccurs="0" maxOccurs="unbounded"
          />
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID"/>
    <xs:attribute name="name" type="xs:string"/>
  </xs:complexType>

</xs:schema>
```

Listing 6.2: Excerpt of the xADL 3.0 Structure Schema as an example for meta model extensions in languages through subclassing[1]

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns="http://www.archstudio.org/xadl3/schemas/implementation-3.0.xsd"
           xmlns:core="http://www.archstudio.org/xadl3/schemas/xadlcore-3.0.xsd"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           attributeFormDefault="qualified"
           elementFormDefault="qualified"
           targetNamespace="http://www.archstudio.org/xadl3/schemas/implementation-3.0
               xsd">

  <xs:import namespace="http://www.archstudio.org/xadl3/schemas/xadlcore-3.0.xsd"
    schemaLocation="https://raw.github.com/isr-uci-edu/ArchStudio5/master/org.
        archstudio.xadl3.xadlcore/model/xadlcore-3.0.xsd"/>

  <xs:complexType abstract="true" name="Implementation">
    <xs:attribute name="id" type="xs:ID"/>
  </xs:complexType>

  <xs:complexType name="ImplementationExtension">
    <xs:complexContent>
      <xs:extension base="core:Extension">
        <xs:sequence>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="implementation" type="
    Implementation"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

Listing 6.3: xADL 3.0 Implementation Schema as an example for meta model extensions in languages through subclassing[2]

[1] Excerpt from: `https://github.com/isr-uci-edu/ArchStudio5/blob/master/org.archstudio.xadl3.structure/model/structure-3.0.xsd`

```
<?xml version="1.0" encoding="UTF-8"?>
<core:xADL xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.archstudio.org/xadl3/schemas/structure-3.0.xsd"
    xmlns:impl="http://www.archstudio.org/xadl3/schemas/implementation-3.0.xsd"
    xmlns:javaimpl="http://www.archstudio.org/xadl3/schemas/javaimplementation-3.0.xsd"
    xmlns:core="http://www.archstudio.org/xadl3/schemas/xadlcore-3.0.xsd">
  <structure id="id1" name="CoCoME">
    <component id="id2" name="CashBox">
      <interface direction="in" id="id3" name="ICashBox"/>
       <impl:implementation xsi:type="javaimpl:JavaImplementation">
          <javaimpl:mainClass javaimpl:className="org.cocome.CashBox"/>
    </component>
    <component id="id4" name="BarcodeScanner">
      <interface direction="out" id="id5" name="IBarcodeScanner"/>
    </component>
    <connector id="id6" name="BarcodeScanner2CashBox">
      <interface direction="in" id="id7" name="IBarcodeScanner"/>
      <interface direction="out" id="id8" name="ICashBox"/>
    </connector>
    <link id="id9" name="[New Link]">
      <point1>id8</point1>
      <point2>id3</point2>
    </link>
    <link id="id10" name="[New Link]">
      <point1>id7</point1>
      <point2>id5</point2>
    </link>
  </structure>
</core:xADL>
```

Listing 6.4: xADL 3.0 Architecture as an example for meta model extensions in languages through subclassing

The meta model extensions strategy allows to extend meta model elements with arbitrary information. In contrast to the untyped placeholders strategy, it implies a structure, that is inherent to the underlying language. Tools can therefore interpret and analyse the architecture, including its extension, as long as they are aware of the extension schemata. The following requirements of Section 6.1 are fulfilled by xADL:

IL-R1 It must be possible to extend the meta model with new first class entities.

It is possible to extend the language with new first class entities, by adding new XML schemata. Adding new first class entities is not intended by xADL. In this language (and its tools) it is only possible to extend classes that provide an extension point.

IL-R2 It must be possible to extend existing elements of the meta model with new properties.

It is possible to extend elements of the language with new properties, by using the extension mechanism. These extensions are not type safe. All extensions (subclasses of `core:Extension`) can be used at an extension point.

IL-R3 It must be possible to model multiple concerns regarding a meta model element simultaneously.

xADL allows for modelling multiple concerns upon an element separately by using the extension mechanism.

[2]Excerpt from: `https://github.com/isr-uci-edu/ArchStudio5/blob/master/org.archstudio.xadl3.implementation/model/implementation-3.0.xsd`

IL-R4 It must be possible to represent components, interfaces, and connectors.

First class entities exist for components, interfaces, and connectors in the *Structure* XML schema.

IL-R5 The meta model must be easily usable with tools working with Ecore meta models.

xADL is based on XML and XML schemata. It is possible to derive Ecore meta models from XML schemata and Ecore models from XML documents. While this is conceptually possible, the technical integration imposes additional challenges.

The meta model extension strategy in general is a good fit for the requirements at hand. The extension mechanism used by xADL, as a representative of architecture languages with meta model extensions, allows for developing meta model extensions for meta model elements, as long as it contains an element of the type *Extension*. It is therefore not possible to develop an extension for components only. There is no type-safety integrated in the extension mechanism. The implementation of this strategy by xADL using extensions with unsafe types implies drawbacks that make the use of xADL unfeasible for the Explicitly Integrated Architecture Process. Also, while it is in theory possible to add new first class entities using XML schemata, this is not intended by the implementation of xADL. Due to the stated drawbacks the strategy is not used for the IAL.

6.2.3 Profiles

Profiles are best known from the context of the UML [Obj15, Section 12.3]. EMF Profiles [LWWC12] is a profile extension for Ecore. With profiles, models and meta models can be extended with additional attributes and references, based on stereotypes. The profiles concept is a special case of the meta model extension strategy. A profile does not introduce first class elements without context, but always extend existing classes with *stereotypes*, which provide another view onto the extended class.

When a stereotype extends a class, the stereotype can be applied to instances of that class. The attributes and references of the stereotype are available to objects that have the stereotype applied, in addition to the attributes and references of the class. Stereotypes are named to make them identifiable. Multiple stereotypes can be applied to a class, therefore providing multiple views onto the same subject.

Profiles are collections of stereotypes, classes, attributes, and references, their interrelationships, and their relationships to abstract syntax elements of other meta models. The meta models that they extend are called base meta models. Profile applications are collections of stereotype applications, further objects and their interrelationships. The relationship between profiles, profile applications, meta models, and models is sketched in Figure 6.2.

Figure 6.3 gives an example of a profile on the basis of Ecore, extended with EMF Profiles. In this example the original meta model consists of one class named *ComponentType*, which has two attributes *name* and *version*, both of the type *String*. A new view upon the class should be established concerning the authorship of component objects. The class is to be extended with an author. This is achieved using a profile. The profile declares a stereotype named *Authored*, which extends the class *ComponentType*. The stereotype declares an attribute *author* of the type *String*. When the *ComponentType* is instantiated, and the stereotype *Authored* is applied to the instance, a value can be assigned to attribute *author* for that specific object. The

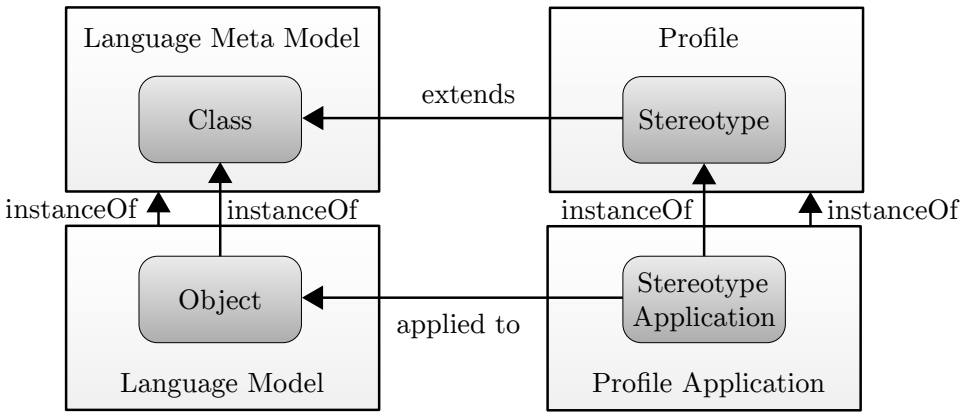


Figure 6.2: The relationships between profiles, profile applications, meta models, and models

stereotype can be applied to an arbitrary number of objects of the class *ComponentType*. The icon to the lower left of the class *ComponentType* declares, that the class is part of a base meta model.

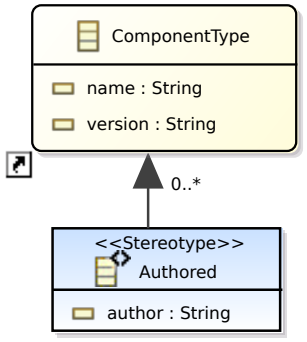


Figure 6.3: Example of a profile with a stereotype, that extends a base meta model class

The profile strategy allows to extend elements with arbitrary information. It is flexible, because arbitrary profiles can be specified, and multiple stereotypes can extend a class. In contrast to the placeholder strategy, the profile strategy uses a well defined structure for that additional information. Other than in the meta model extension strategy, the extensions with stereotypes are type safe, because it is explicitly declared which class can be extended with a specific stereotype. Profiles can also be a base meta model of other profiles. Therefore profiles can build upon each other. A stereotype can extend classes, that are part of any base meta model. This strategy fulfills the requirements as follows:

IL-R1 It must be possible to extend the meta model with new first class entities.

The use of the profile strategy allows for adding arbitrary abstract syntax elements to a meta model. A profile can define new classes, which can be contained by a stereotype. Therefore it is possible to integrate new architectural concepts into an architecture language defined with profiles.

IL-R2 It must be possible to extend existing elements of the meta model with new properties.

It is possible to extend meta model elements with new properties using the profile strategy. By declaring and applying stereotypes, new attributes and references can be added to existing meta model elements.

IL-R3 It must be possible to model multiple concerns regarding a meta model element simultaneously.

The profile strategy allows for modelling multiple concerns regarding a model element simultaneously, by extending a class with multiple stereotypes and applying multiple stereotypes to an object.

IL-R4 It must be possible to represent components, interfaces, and connectors.

The strategy and EMF Profiles as tool, that implements the strategy, do not fulfil this requirement for themselves. To fulfill this requirement, a meta model or profiles have to be created, that allow for modelling these elements.

IL-R5 The meta model must be easily usable with tools working with Ecore meta models.

The profile mechanism can be used with Ecore using EMF Profiles. EMF Profiles is designed to be usable with the existing ecosystem for Ecore meta models and models.

Four of five requirements are fulfilled by choosing profiles as strategy for extending a meta model. The remaining requirement can be fulfilled by creating a meta model and profiles for representing the respective elements. This strategy has been chosen for the Explicitly Integrated Architecture approach. For using the profiles strategy, it is necessary to define how profiles integrate with the formalization of meta models and models. In the following sections, we define how stereotypes are used within the meta model definition of Section 5.4.1.

Stereotypes

Stereotypes are named elements that may have attributes and references. Stereotypes extend classes.

Definition 74: Stereotypes as Named Elements

A stereotype is a named element. The naming is defined analogously to the Definition 3 of named elements in meta models. For a set $Stereotypes$ of stereotypes and a set N of labels, the function $name$ that assigns a name to a stereotype is defined as follows:

$$name : Stereotypes \rightarrow N$$

Definition 75: Stereotypes own Attributes and References

A stereotype has attributes and references. This is defined analogously to Definition 4 of the respective ownerships for classes in meta models. For a set $Stereotypes$ of stereotypes, a set $Attributes$ of attributes and a set $References$ of references, we define the relation $\xrightarrow{has}$, that declares that a stereotype has the given attribute or reference.

$$\xrightarrow{has} \subseteq Stereotypes \times (Attributes \cup References)$$

The shorthand notation for ownership of Definition 4 also applies to stereotypes and their attributes and references. For a stereotype s and an attribute or reference $e \in Attributes \cup References$:

$$s.e : \Longleftrightarrow e, s \xrightarrow{has} e$$

Instead of the elements, their name can also be used. E.g. for a stereotype $s \in Stereotypes$ and an attribute $a \in Attributes$:

$$s \xrightarrow{has} a \wedge name(s) = \text{Authored} \wedge name(a) = \text{name} \Leftrightarrow \text{Authored.name} = a$$

Definition 76: Stereotypes extend Classes

A stereotype extends classes. For a set *Stereotypes* of stereotypes and a set *Classes* of classes, we define the relation $\xrightarrow{extends}$ that declares that a stereotype extends a class.

$$\xrightarrow{extends} \subseteq Stereotypes \times Classes$$

A stereotype can be applied to an object, when the corresponding stereotype extends the object's class. When a stereotype is applied to an object, values can be assigned to the attributes and references of the stereotype for that object.

Definition 77: Stereotypes Applied to Objects

The application of a stereotype to an object is defined with a relation $\xrightarrow{appliedTo}$.

$$\xrightarrow{appliedTo} \subseteq Stereotypes \times O$$

Constraint 15: Constraints for Applying Stereotypes to Objects

A stereotype can only be applied to an object, when the stereotype extends the object's class.

Let s be a stereotype and o be an object.

$$s \xrightarrow{appliedTo} o \implies \exists c \in Classes : o \xrightarrow{instanceOf} c \wedge s \xrightarrow{extends} c$$

Constraint 16: Assigning Values to Attributes of Stereotypes

When a stereotype is applied to an object, values can be assigned to the stereotype's attributes for that object. In the context of profile applications, Definition 12 for assigning values to attributes is weakened as follows:

$$(o, a) \xrightarrow{has} v \implies (\exists c \in Classes : o \xrightarrow{instanceOf} c \wedge c \xrightarrow{has} a)$$

$$\vee\ (\exists s \in Stereotypes : s \xrightarrow{appliedTo} o \wedge s \xrightarrow{has} a)$$

Constraint 17: Assigning Targets to References of Stereotypes

When a stereotype is applied to an object, targets can be assigned to the stereotype's references for that object. In the context of profile applications, Definition 13 for assigning targets to references is weakened as follows:

$$(o_{source}, r) \xrightarrow{references} o_{target} \implies (\exists c \in Classes : o_{source} \xrightarrow{instanceOf} c \wedge c \xrightarrow{has} r)$$
$$\vee\ (\exists s \in Stereotypes : s \xrightarrow{appliedTo} o \wedge s \xrightarrow{has} r)$$

Profiles

Profiles are collections of stereotypes, classes, attributes, and references that have a common concern. They are applied to a one or more base meta models, meaning that their stereotypes can extend classes of the base meta model, their references can target classes of the base meta model, and their attributes can be typed by the data types of the base meta models. Profiles are modelling language meta models themselves, and can be applied to other profiles.

The definition of a profile's relations and functions is close to the corresponding definition in modelling language meta models (see Definition 1). In comparison to the modelling language meta model definition, references within profiles can also target abstract syntax elements of the base meta models, the type of attributes within profiles can be part of the base meta models, and stereotypes extend classes of the base meta models.

Definition 78: Profiles

A profile $Profile$ is a meta model (see Definition 1) extended with stereotypes.

$$Profile := (B, A, D, L, F), \text{ where}$$

- B is a set of base meta models (possibly including profiles).
- A, D, and L are defined analogously to their definition for modelling language meta models (see Definition 1)
- the typing function

  $$\xrightarrow{isOf}: A \to \{StereotypeType, ClassType, AttributeType, ReferenceType\}$$

  considers Stereotypes in addition to the types of abstract syntax elements in meta models.
- the set of relations and functions F is based on those of modelling language meta models, but extended for profiles in the following definitions.

$Stereotypes_{Profile}$ is the set of all stereotypes in the profile $Profile$.

$$Stereotypes_{Profile} := \{a \mid a \in A_{Profile} \wedge a \xrightarrow{isOf} StereotypeType\}$$

Definition 79: Relationship between Abstract Syntax Elements of Profiles and Modelling Language Meta Models

Let $Classes_{Profile}$ be the set of classes in a profile, and $Classes_B$ be the set of classes in the abstract syntax elements of the profile's base meta models. Further let $D_{Profile}$ be the set of data types in the profile, and D_B the set of data types in the profile's base meta models. Then

- $\xrightarrow{extends}$: $Stereotypes \rightarrow Classes_{Profile} \cup Classes_B$ defines that a stereotype extends a class of the profile or the base meta model.
- $\xrightarrow{isOfType}$: $References \rightarrow Classes_{Profile} \cup Classes_B$, defines that a reference of the profile has a class of the profile or a base meta model as type.[a]
- $\xrightarrow{isOfType}$: $Attributes \rightarrow D_{Profiles} \cup D_B$ defines that an attribute has a data type of the profile or a base meta model as type.

[a] All relations $x \xrightarrow{isOfType} y$ can also be notated as $type(x) = y$.

Definition 80: Profile Application

A profile application is an instance of a profile, like a model is an instance of a meta model. It is defined as a tuple

$$PA := (P, O, V, N, F, R), \text{ where}$$

- P is the profile that the application instantiates,
- and O, V, N, F, and R are defined analogously their definition of models (see Definition 9).

For the set of profile applications $ProfileApplications$ and the set of profiles $Profiles$, the function $\xrightarrow{instanceOf}$ declares that a profile application instantiates a profile.

$$\xrightarrow{instanceOf}: ProfileApplications \times Profiles$$

Example

Example 7 shows how profiles and stereotypes are used. This formal description represents the example given in Figure 6.3. It declares a class *ComponentType*. The profile declares a stereotype *Authored*, that extends *ComponentType* and adds an attribute *author*.

Example 8 shows how profile applications are used. The base model $M^{Example}$ is defined on Example 2. The profile application is depicted in Figure 6.4. It applies the stereotype *Authored* to the two objects of the class *ComponentType*.

Example 7: Profile Example

The formalization of the profile $Profile^{Example}$, represented in Figure 6.3, is defined as follows. Empty sets are not explicitly stated. The base meta model $M_{Meta}^{Example}$ is defined in Example 1.

$$B := \{M_{Meta}^{Example}\}, \qquad Stereotypes := \{\text{s}\}, \qquad Attributes := \{\text{a}\}$$

The elements are named as follows:

$$name(\text{s}) = \text{Authored}, \qquad name(\text{a}) = \text{author}$$

The following relations apply:

$$\text{Authored.author} \xrightarrow{isOfType} String, \qquad \text{Authored.author} \xrightarrow{extends} \text{ComponentType}$$

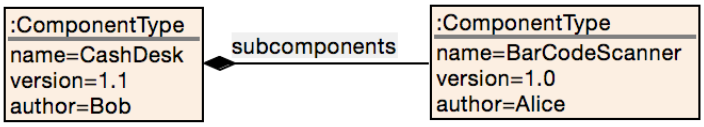


Figure 6.4: A model with the example profile application

Example 8: Profile Application Example

The example profile application $PA^{Example} \xrightarrow{instanceOf} Profile^{Example}$ is formalized as follows. Empty sets are not explicitly stated. The referenced $M^{Example}$ is defined in Example 2.

$$P := Profile^{Example}, \qquad R := \{M^{Example}\}$$

The following relations apply:

$$\text{barCodeScanner.author} \xrightarrow{hasValue} \text{Alice}, \qquad \text{cashDesk.author} \xrightarrow{hasValue} \text{Bob}$$

6.2.4 Language Management

In addition to the strategies described above, a language with extensions should be subject of a management task. The existence of language extensions should be registered and described, so that all language users can have an overview of the possible expressiveness, including which extensions exclude or require each other.

As an example for mutual exclusion, consider the Palladio Component Model (PCM) [BKR09], which describes hierarchical component instantiation, while Enterprise Java Beans (EJB) [EJB09] describes a flat component instantiation. When an intermediate

language wants to describe such architectures, it must ensure that the architecture is not invalid, when it is declared flat and hierarchical at the same time. An example for mutual requirement can be event-based interfaces and event-based connectors. Event-based connectors are only useful when event-based interfaces are present. Language management declares such exclusions and requirements. When a model is used, a valid subset of the model has to be interpreted.

6.3 Meta Model Overview

The Explicitly Integrated Architecture Process uses the general purpose architecture language Intermediate Architecture Description Language (IAL) to translate between architecture implementation and specification languages. The IAL comprises a language kernel extended with profiles. The kernel defines the common meta model elements of all architectures, that are described with the IAL. The profiles extend the kernel with meta model elements of specific concerns. The kernel is described in Section 6.4. Section 6.5 describes the profiles and their interrelationships.

6.4 Language Kernel

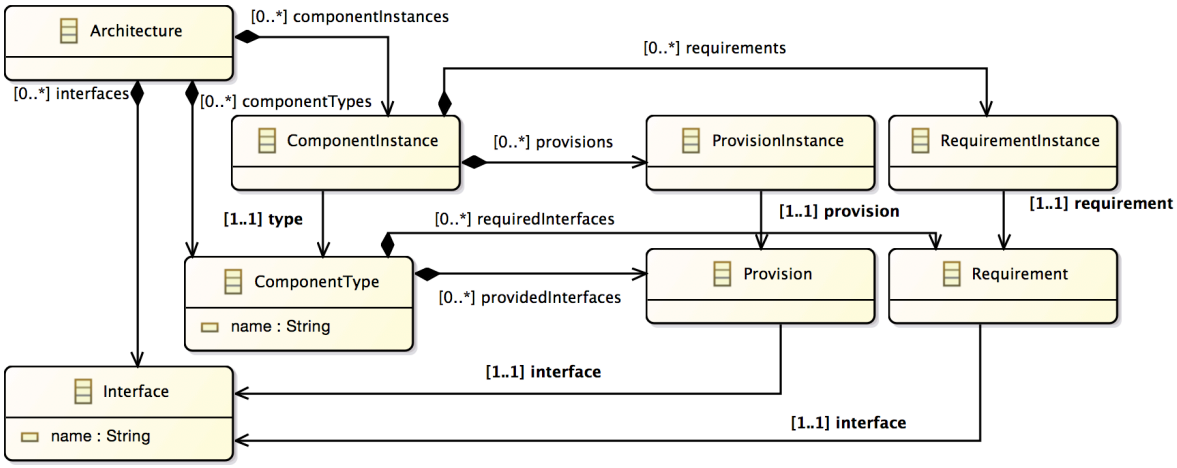


Figure 6.5: The kernel of the Intermediate Architecture Description Language

Figure 6.5 shows the kernel of the IAL. The *Architecture* is the root node that represents a software architecture comprising interconnected components. The class *ComponentType* represents a static design-time component description. The component type has an attribute *name* for declaring a component type name. The class *Interface* can be used as an abstract definition of an interface for a component. It does not imply a structure or a specific behaviour of the interface. It can therefore represent different styles of interfaces, e.g. a procedure call interface or an event-based interface. Interfaces are named. Component types can provide and require interfaces via the *Provision* and *Requirement* elements.

The class *Component Instance* represents the runtime view on components types. When a component type is executed, a component instance is created. The provision and requirement of interfaces are instantiated respectively. At design time, instances can be representatives of several instances (see Section 6.5.4).

The language kernel is formalized as a modelling language meta model as follows:

Definition 81: Intermediate Architecture Description Language Kernel

The meta model of the Intermediate Architecture Description Language M_{Meta}^{Kernel} is defined as follows:

$$Classes := \{c^{Arch}, c^{Int}, c^{CT}, c^{CI}, c^{Req}, c^{Prov}, c^{ReqI}, c^{ProvI}\}$$

$$Attributes := \{a_{name}^{int}, a_{name}^{ct}\}$$

$$References := \{r^{arch2ci}, r^{arch2ct}, r^{arch2int}, r^{ci2ct}, r^{ci2provi}, r^{ci2reqi}, r^{ct2req}, r^{ct2prov}, r^{prov2int}, r^{provi2prov}, r^{req2int}, r^{reqi2req}\}$$

$$Containment := \{r^{arch2ci}, r^{arch2ct}, r^{arch2int}, r^{ci2provi}, r^{ci2reqi}, r^{ct2req}, r^{ct2prov}\}$$

The elements are named as follows:

$name(c^{Arch})$ = Architecture, $name(c^{CI})$ = ComponentInstance,
$name(c^{CT})$ = ComponentType, $name(c^{Int})$ = Interface,
$name(c^{Prov})$ = Provision, $name(c^{ProvI})$ = ProvisionInstance,
$name(c^{Req})$ = Requirement, $name(c^{ReqI})$ = RequirementInstance,
$name(a_{name}^{int})$ = name, $name(a_{name}^{ct})$ = name,
$name(r^{arch2ci})$ = componentInstances, $name(r^{arch2ct})$ = componentTypes,
$name(r^{arch2int})$ = interfaces, $name(r^{ci2provi})$ = provisions,
$name(r^{ci2reqi})$ = requirements, $name(r^{ci2ct})$ = type,
$name(r^{ct2req})$ = requiredInterfaces, $name(r^{ct2prov})$ = providedInterfaces,
$name(r^{prov2int})$ = interface, $name(r^{provi2prov})$ = provision,
$name(r^{req2int})$ = interface, $name(r^{reqi2req})$ = requirement

The attributes and references are defined as follows:

$$Architecture.componentInstances \xrightarrow{isOfType} ComponentInstance,$$
$$Architecture.componentTypes \xrightarrow{isOfType} ComponentType,$$
$$Architecture.interfaces \xrightarrow{isOfType} Interface,$$

$$
\begin{aligned}
ComponentInstance.provisions &\xrightarrow{isOfType} ProvisionInstance,\\
ComponentInstance.requirements &\xrightarrow{isOfType} RequirementInstance,\\
ComponentInstance.type &\xrightarrow{isOfType} ComponentType,\\
ComponentInstance.type &\xrightarrow{cardinality} 1..1,\\
ComponentType &\xrightarrow{has} a^{ct}_{name},\\
ComponentType.name &\xrightarrow{isOfType} String,\\
ComponentType.providedInterfaces &\xrightarrow{isOfType} Provision,\\
ComponentType.requiredInterfaces &\xrightarrow{isOfType} Requirement,\\
Interface &\xrightarrow{has} a^{int}_{name},\\
Interface.name &\xrightarrow{isOfType} String,\\
Provision.interface &\xrightarrow{isOfType} Interface,\\
Provision.interface &\xrightarrow{cardinality} 1..1,\\
ProvisionInstance.provision &\xrightarrow{isOfType} Provision,\\
ProvisionInstance.provision &\xrightarrow{cardinality} 1..1,\\
Requirement.interface &\xrightarrow{isOfType} Interface,\\
Requirement.interface &\xrightarrow{cardinality} 1..1,\\
RequirementInstance.requirement &\xrightarrow{isOfType} Requirement,\\
RequirementInstance.requirement &\xrightarrow{cardinality} 1..1
\end{aligned}
$$

6.5 Language Profiles

An architecture modelled with the IAL comprises elements of the kernel and elements of profiles. The kernel includes the core aspects of architectural descriptions in the terms of this thesis – components and interfaces. Profiles add further concerns to the architecture language. Such concerns include e.g. different types of connectors, component hierarchies, types of interfaces, or quality aspects.

Architecture models of the IAL may include inconsistent information. E.g. an architecture can include information about a deep component hierarchy, and at the same time include the information that the architecture is flat. The semantics of this model is based on the interpreter. Interpreting a profile is optional within the boundaries of exclusions and requirements between profiles. Therefore in the example given above, an architecture can be interpreted as a deep component type hierarchy when an interpreter handles deep hierarchies. When the interpreter can only handle flat hierarchies, the profile for deep hierarchies can be ignored, and the profile for flat hierarchies can be used as the source of information. The architecture is now interpreted, and possibly changed with a view on a flat hierarchy of component types, without loosing the information about the actual hierarchy. Profiles are not always independent, but can require

or exclude each other. E.g. when a profile declaring event-based connectors is interpreted, it might be require to also interpret event-based interfaces. When a component hierarchy is interpreted to be flat, it cannot at the same time be interpreted as deep.

Profiles can be categorized regarding their abstract concern. E.g. the profiles *Flat Component Hierarchy*, and *Scoped Component Hierarchy* both handle the abstract concern of the component hierarchy, or *Time Resource Demand* and *Security Levels* both handle software quality concerns. Some categories are mandatory, meaning that at least one profile has to be used when an architecture is described. One kind of component type hierarchy must be chosen. Some categories contain only optional profiles. E.g. no software quality profile is necessary to be used.

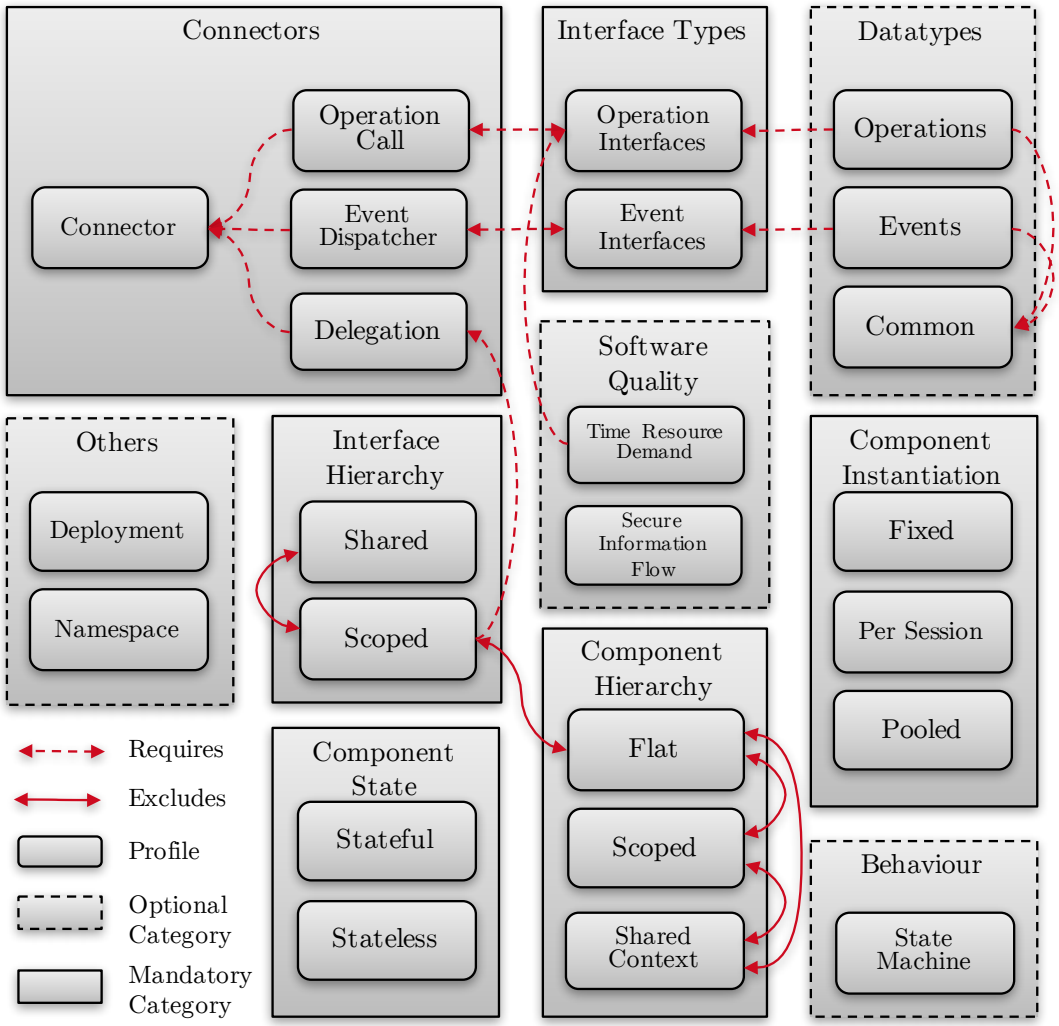


Figure 6.6: An overview of profiles of the Intermediate Architecture Description Language and their interrelationships

Figure 6.6 shows the profiles of the IAL, and their interrelationships regarding their inter-

pretation. The rectangles are categories of profiles, which share an abstract concern. The rectangles with rounded corners represent profiles. Mandatory categories (which have a solid border in Figure 6.6) require at least one profile to be used. Arrows between profiles show whether the interpretation of a profile requires or excludes the interpretation of another profile. An arrow with a dashed line defines that when the source of the arrow is interpreted, it is required that its target is also interpreted. An arrow with a solid line defines that when the source of the arrow is interpreted, it is excluded that its target is interpreted. The reasons for these requirements and exclusions is stated in the sections describing the corresponding profiles. The following sections describe and formalize the current profiles and categories in the Intermediate Architecture Description Language. An example for the application of each profile is given in Appendix C. Considering the objective of the language, in the future more profiles and categories can be added.

6.5.1 Interface Types

In the language kernel the interface definition is of an abstract nature. The following profiles describe different types of interfaces.

Operation Interfaces

The Operation Interfaces profile (see Figure 6.7) declares operations that can be invoked by their clients. Operations have parameters and return types. Operations and operation parameters are named. Operation interfaces are very common in software architectures, both in architecture implementation and architecture specification languages. Examples are Session Beans in Enterprise JavaBeans (EJB) [EJB13], where beans invoke each other's operations, or the Palladio Component Model [BKR09], an architecture specification language, which has operations in interfaces as entities and behaviour specifications that declare operation invocations.

The *OperationInterface* stereotype can be applied to an interface to give it the role of an operation interface. This profile requires the use of the Operation Call Connector profile (see Definition 96) for connecting the requirements and provisions of operation-based interfaces.

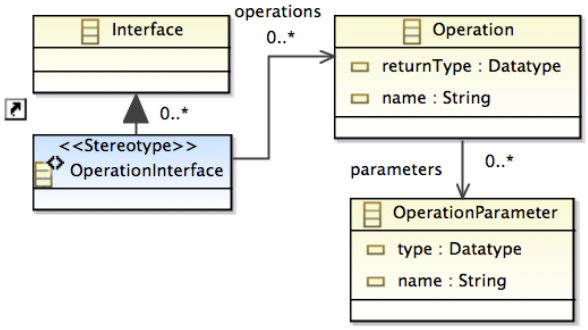


Figure 6.7: The *Operation Interfaces* profile of the IAL

Formalization Definition 82 gives a formal description of the profile.

Definition 82: Interface Type Operations Profile

The Interface Type Operations profile $P_{OperationInterfaces}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}\}$$

$$Stereotypes := \{s_{OpInt}\}$$

$$Classes := \{c_{Op}, c_{OpPar}\}$$

$$Attributes := \{a_{Op\_retTyp}, a_{Op\_name}, a_{OpPar\_typ}, a_{OpPar\_name}\}$$

$$References := \{r_{OpInt\_op}, r_{Op\_par}\}$$

$$Containments := \{r_{OpInt\_op}, r_{Op\_par}\}$$

The elements are named as follows:

$$\begin{aligned} name(s_{OpInt}) &= \text{OperationInterface}, \\ name(c_{Op}) &= \text{Operation}, \\ name(c_{OpPar}) &= \text{OperationParameter}, \\ name(a_{Op\_retTyp}) &= \text{returnType}, \\ name(a_{Op\_name}) &= \text{name}, \\ name(a_{OpPar\_typ}) &= \text{type}, \\ name(a_{OpPar\_name}) &= \text{name}, \\ name(r_{OpInt\_op}) &= \text{operations}, \end{aligned}$$

$$name(r_{Op\_par}) = \text{parameters}$$

The stereotypes extend the following classes:

$$\text{OperationInterface} \xrightarrow{extends} \text{Interface}$$

The attributes and references are defined as follows:

$$\begin{aligned}
\text{Operation.returnType} &\xrightarrow{isOfType} \text{Datatype},\\
\text{Operation.name} &\xrightarrow{isOfType} \text{String},\\
\text{OperationParameter.type} &\xrightarrow{isOfType} \text{Datatype},\\
\text{OperationParameter.name} &\xrightarrow{isOfType} \text{String},\\
\text{OperationInterface.operations} &\xrightarrow{isOfType} \text{Operation},\\
\text{Operation.parameters} &\xrightarrow{isOfType} \text{OperationParameter}
\end{aligned}$$

Event Interfaces

The Event Interfaces profile (see Figure 6.8) declare events and event parameters. Components that provide an event-based interface reacts on these events. Components requiring an event-based interface triggers these events. Events and their parameters are named. Event parameters are typed. Just like operation interfaces, event interfaces are common in software architectures, both in architecture implementation and architecture specification languages. Examples are Message-Driven Beans in EJB, where beans invoke each other's operations, or the Palladio Component Model, an architecture specification language, which has operations in interfaces as entities and behaviour specifications that declare operation invocations.

The *EventInterface* stereotype can be applied to an interface to give it the role of an event-based interface. This profile requires the use of the Event Dispatcher Connector profile for connecting the requirements and provisions of event-based interfaces.

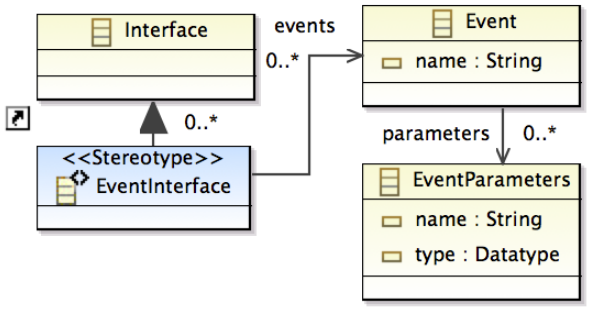


Figure 6.8: The *Event Interfaces* profile of the IAL

Formalization Definition 83 gives a formal description of the profile.

Definition 83: Interface Type Events Profile

The Interface Type Events profile $P_{EventInterfaces}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}\}$$
$$Stereotypes := \{s_{EvInt}\}$$
$$Classes := \{c_{Ev}, c_{EvPar}\}$$
$$Attributes := \{a_{Ev\_name}, a_{EvPar\_name}, a_{EvPar\_typ}\}$$
$$References := \{r_{EvInt\_ev}, r_{Ev\_par}\}$$
$$Containments := \{r_{EvInt\_ev}, r_{Ev\_par}\}$$

The elements are named as follows:

$$\begin{aligned} name(s_{EvInt}) &= \text{EventInterface}, \\ name(c_{Ev}) &= \text{Event}, \\ name(c_{EvPar}) &= \text{EventParameters}, \\ name(a_{Ev\_name}) &= \text{name}, \\ name(a_{EvPar\_name}) &= \text{name}, \\ name(a_{EvPar\_typ}) &= \text{type}, \\ name(r_{EvInt\_ev}) &= \text{events}, \\ name(r_{Ev\_par}) &= \text{parameters} \end{aligned}$$

The stereotypes extend the following classes:

$$\text{EventInterface} \xrightarrow{extends} \text{Interface}$$

The attributes and references are defined as follows:

$$\begin{aligned} \text{Event.name} &\xrightarrow{isOfType} \text{String}, \\ \text{EventParameters.name} &\xrightarrow{isOfType} \text{String}, \\ \text{EventParameters.type} &\xrightarrow{isOfType} \text{Datatype}, \\ \text{EventInterface.events} &\xrightarrow{isOfType} \text{Event}, \\ \text{Event.parameters} &\xrightarrow{isOfType} \text{EventParameters} \end{aligned}$$

6.5.2 Interface Hierarchy

These profiles consider where interfaces are declared within the architecture. The interpretation of these profiles exclude each other, so that an interface hierarchy can only be interpreted as either shared interface hierarchy, or scoped interface hierarchy.

Shared Interface Hierarchy

In the Shared Interface Hierarchy profile (see Figure 6.9), all interfaces are declared within the same scope, so that there are no visibility constraints. Shared interface hierarchies are known e.g. from the Java Enterprise Edition, where no kind of component or interface hierarchy is defined. As an example for specification languages, the PCM uses a shared repository, where all interfaces of a system are stored.

The stereotype *SharedInterfacesArchitecture* can be applied to an architecture to signal that this hierarchy type is modelled. The kernel already declares the *Interface* class as a child of the *Architecture* class. Therefore no more information is necessary.

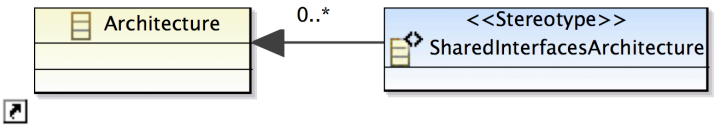


Figure 6.9: The *Shared Interface Hierarchy* profile of the IAL

Formalization Definition 84 gives a formal description of the profile.

Definition 84: Interface Hierarchy Shared Profile

The Interface Hierarchy Shared profile $P_{SharedInterfaceHierarchy}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}\}$$

$$Stereotypes := \{s_{SharIntAr}\}$$

The elements are named as follows:

$$name(s_{SharIntAr}) = \text{SharedInterfacesArchitecture}$$

The stereotypes extend the following classes:

$$\text{SharedInterfacesArchitecture} \xrightarrow{extends} \text{Architecture}$$

Scoped Interface Hierarchy

In the Scoped Interface Hierarchy profile (see Figure 6.10), each interface is declared within the scope of a component type or within the scope of the architecture. This means that the interface can only be required or provided by component types that are children of this component type declaration or architecture and – when there is a deeper hierarchy of component types – within deeper levels. The stereotype *ScopedInterfacesArchitecture* can be applied to an architecture to signal that this hierarchy type is modelled. Additionally, the stereotype *ScopedInterfacesComponentType* can be applied to a component type to add information about

the scope of an interface. A *ScopedInterfacesComponentType* references interfaces that are declared within its component type's scope. Scoped interface hierarchies are used primarily in architecture specification languages. E.g. in the UML [Obj15] components can declare interfaces as so-called *packagedElements* within their scope.

This profile requires the use of the Delegation Connector profile (see Definition 98), for forwarding provisions and requirements of interfaces to subcomponents. This profile excludes the Flat Component Hierarchy profile (see Definition 86). It declares interfaces in the scope of component types, and these interfaces can only be provided or required by component types declared within the same scoped. It would not be possible for a component type to provide or require a child interface of another component type in a flat component type hierarchy.

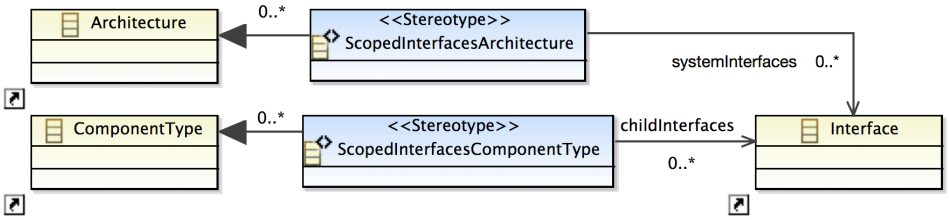


Figure 6.10: The *Scoped Interface Hierarchy* profile of the IAL

Formalization Definition 85 gives a formal description of the profile.

Definition 85: Interface Hierarchy Scoped Profile

The Interface Hierarchy Scoped profile $P_{ScopedInterfaceHierarchy}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}\}$$

$$Stereotypes := \{s_{ScIntCompTyp}, s_{ScIntAr}\}$$

$$References := \{r_{ScIntCompTyp\_chilInt}, r_{ScIntAr\_systInt}\}$$

The elements are named as follows:

$$\begin{aligned} name(s_{ScIntCompTyp}) &= \text{ScopedInterfacesComponentType}, \\ name(s_{ScIntAr}) &= \text{ScopedInterfacesArchitecture}, \\ name(r_{ScIntCompTyp\_chilInt}) &= \text{childInterfaces}, \\ name(r_{ScIntAr\_systInt}) &= \text{systemInterfaces} \end{aligned}$$

The stereotypes extend the following classes:

$$\begin{aligned} \text{ScopedInterfacesComponentType} &\xrightarrow{extends} \text{ComponentType}, \\ \text{ScopedInterfacesArchitecture} &\xrightarrow{extends} \text{Architecture} \end{aligned}$$

The attributes and references are defined as follows:

$$\text{ScopedInterfacesComponentType.childInterfaces} \xrightarrow{isOfType} \text{Interface},$$
$$\text{ScopedInterfacesArchitecture.systemInterfaces} \xrightarrow{isOfType} \text{Interface}$$

6.5.3 Component Hierarchy

These profiles consider the type of hierarchy for component types. The interpretation of these profiles exclude each other, so that a component hierarchy can only be interpreted as either flat, shared, or scoped.

Flat Component Hierarchy

In this profile (see Figure 6.11) component types and their instances are all located in the same scope. Flat component hierarchies are rather common in architecture implementation languages. Examples for such languages used broadly in practice are the JEE or OSGi [The14].

The stereotype *HierarchicalArchitectureFlat* can be applied to an architecture to denote a flat component hierarchy. This profile excludes the Scoped Interface Hierarchy profile (see Definition 85). The profile Scoped Interface Hierarchy declares interfaces in the scope of component types, and these interfaces can only be provided or required by component types declared within the same scope. It would not be possible for a component type to provide or require a child interface of another component type in a flat component type hierarchy.

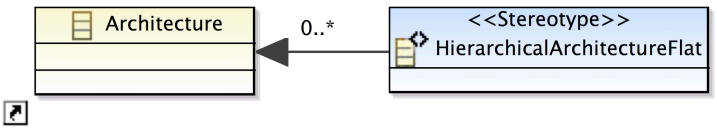


Figure 6.11: The *Flat Component Hierarchy* profile of the IAL

Formalization Definition 86 gives a formal description of the profile.

Definition 86: Component Hierarchy Flat Profile

The Component Hierarchy Flat profile $P_{FlatComponentHierarchy}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}\}$$
$$Stereotypes := \{s_{HierArFl}\}$$

The elements are named as follows:

$$name(s_{HierArFl}) = \text{HierarchicalArchitectureFlat}$$

The stereotypes extend the following classes:

$$\text{HierarchicalArchitectureFlat} \xrightarrow{extends} \text{Architecture}$$

Scoped Component Hierarchy

The Scoped Component Hierarchy profile (see Figure 6.12) describes a hierarchical component type architecture. The architecture and each component type are a scope, under which component instances and other component types can be declared. Scoped component hierarchies are commonly provided in architecture specification languages. As an example, UML provides scoped component hierarchies.

The stereotype *HierarchicalArchitectureScoped* can be applied to an architecture to signal that a scoped component hierarchy is modelled. An architecture has at least one component type as system type. The stereotype *HierarchicalComponentTypeScoped* can be applied to component types. These component types can reference other component types as child types and instances of these types as child instances. System instances are component instances in the top level scope.

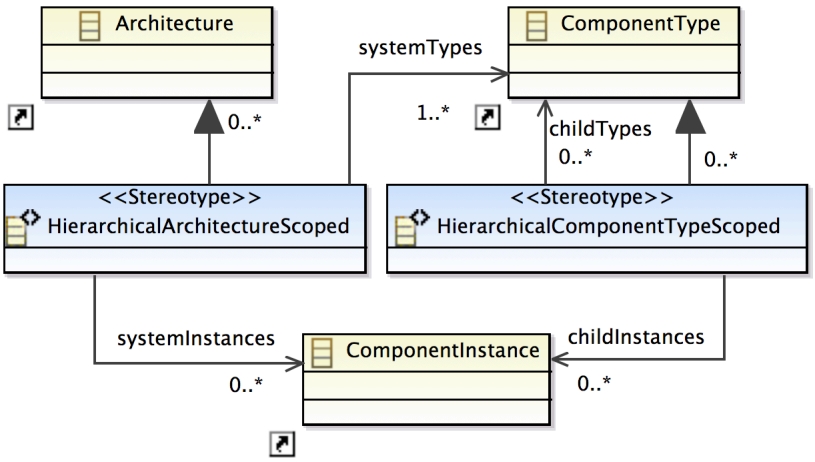


Figure 6.12: The *Scoped Component Hierarchy* profile of the IAL

Formalization Definition 87 gives a formal description of the profile.

Definition 87: Component Hierarchy Scoped Profile

The Component Hierarchy Scoped profile $P_{ScopedComponentHierarchy}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}\}$$

$$Stereotypes := \{s_{HierCompTypSc}, s_{HierArSc}\}$$

$$References := \{r_{HierCompTypSc\_chilInst}, r_{HierCompTypSc\_chilTyp}, r_{HierArSc\_systTyp}, r_{HierArSc\_systInst}\}$$

The elements are named as follows:

$$\begin{aligned} name(s_{HierCompTypSc}) &= \text{HierarchicalComponentTypeScoped}, \\ name(s_{HierArSc}) &= \text{HierarchicalArchitectureScoped}, \\ name(r_{HierCompTypSc\_chilInst}) &= \text{childInstances}, \\ name(r_{HierCompTypSc\_chilTyp}) &= \text{childTypes}, \\ name(r_{HierArSc\_systTyp}) &= \text{systemTypes}, \\ name(r_{HierArSc\_systInst}) &= \text{systemInstances} \end{aligned}$$

The stereotypes extend the following classes:

$$\begin{aligned} \text{HierarchicalComponentTypeScoped} &\xrightarrow{extends} \text{ComponentType}, \\ \text{HierarchicalArchitectureScoped} &\xrightarrow{extends} \text{Architecture} \end{aligned}$$

The attributes and references are defined as follows:

$$\begin{aligned} \text{HierarchicalComponentTypeScoped.childInstances} &\xrightarrow{isOfType} \text{ComponentInstance}, \\ \text{HierarchicalComponentTypeScoped.childTypes} &\xrightarrow{isOfType} \text{ComponentType}, \\ \text{HierarchicalArchitectureScoped.systemTypes} &\xrightarrow{cardinality} 1..*, \\ \text{HierarchicalArchitectureScoped.systemTypes} &\xrightarrow{isOfType} \text{ComponentType}, \\ \text{HierarchicalArchitectureScoped.systemInstances} &\xrightarrow{isOfType} \text{ComponentInstance} \end{aligned}$$

Shared Context Component Hierarchy

In this profile (see Figure 6.13), component types are all declared within the same scope. Component types in this profile can be composed by child component instances. When a parent component type is instantiated in the running system, its child instances are also created. Therefore a hierarchy of component instances is modelled. The PCM is an example for an architecture specification language with a shared context component hierarchy.

The stereotype *HierarchicalArchitectureSharedContext* can be applied to an architecture to signal that a shared context component hierarchy is modelled. Such an architecture declares system instances, i.e. component instances that exist when the system is instantiated. The stereotype *HierarchicalComponentTypeSharedContext* can be applied to a component type. Such a component type can reference component instances as child instances.

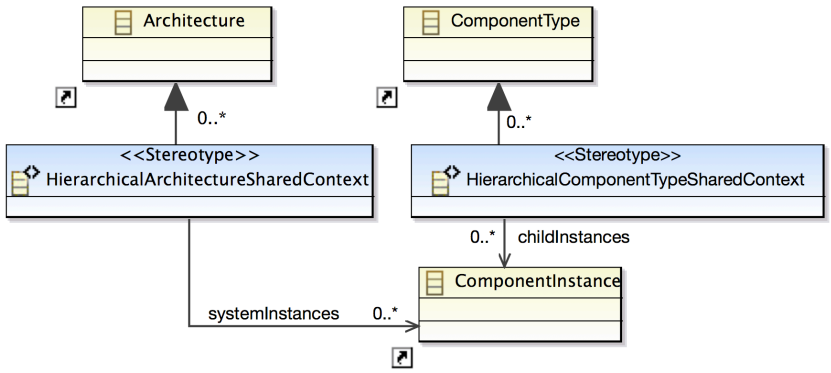


Figure 6.13: The *Shared Context Component Hierarchy* profile of the IAL

Formalization Definition 88 gives a formal description of the profile.

Definition 88: Component Hierarchy Shared Profile

The Component Hierarchy Shared profile $P_{SharedContextComponentHierarchy}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}\}$$

$$Stereotypes := \{s_{HierArSharCont}, s_{HierCompTypSharCont}\}$$

$$References := \{r_{HierArSharCont\_systInst}, r_{HierCompTypSharCont\_chilInst}\}$$

The elements are named as follows:

$$\begin{aligned} name(s_{HierArSharCont}) &= \text{HierarchicalArchitectureSharedContext}, \\ name(s_{HierCompTypSharCont}) &= \text{HierarchicalComponentTypeSharedContext}, \\ name(r_{HierArSharCont\_systInst}) &= \text{systemInstances}, \\ name(r_{HierCompTypSharCont\_chilInst}) &= \text{childInstances} \end{aligned}$$

The stereotypes extend the following classes:

$$\begin{aligned} \text{HierarchicalArchitectureSharedContext} &\xrightarrow{extends} \text{Architecture}, \\ \text{HierarchicalComponentTypeSharedContext} &\xrightarrow{extends} \text{ComponentType} \end{aligned}$$

The attributes and references are defined as follows:

$$\begin{aligned} &\text{HierarchicalArchitectureSharedContext.systemInstances} \xrightarrow{isOfType} \text{ComponentInstance}, \\ &\text{HierarchicalComponentTypeSharedContext.childInstances} \xrightarrow{isOfType} \text{ComponentInstance} \end{aligned}$$

6.5.4 Component Instantiation

When a component-based software is executed, the component types are instantiated. I.e. executable units are created that represent the modelled functionality at run time. How component types are instantiated varies. For some component types, a statically defined fix number of instances are created. For other component types the number of instances is dynamically chosen, e.g. one instance per user session or the number of instances is based on the load, so when the load increases new instances can be created to cover the load. The static representation of how component types are instantiated is modelled using the following profiles. These profiles can be used simultaneously in an architecture, albeit it is not allowed to declare multiple of these stereotypes on the same component type.

Fixed Component Instantiation

In the Fixed Component Instantiation profile (see Figure 6.14), the amount of instances of a component type is declared statically in the model, and cannot change at run time. When the component type is instantiated that exact number of instances must be created. Declaring component instances explicitly seems to be more common in architecture specification languages, than in architecture implementation languages. E.g. in PCM or the UML, a fixed number of instances can be declared. In architecture implementation languages it is sometimes possible to declare singleton component types. These types are instantiated exactly once. This is a special case of a fixed number of component instances. Examples for this kind of instantiation are application scoped beans in CDI. Architecture languages that allow for declaring component instances explicitly seem to be more common To model a fixed component instantiation, the stereotype *ComponentInstancesFixedType* has to be applied to a component type.

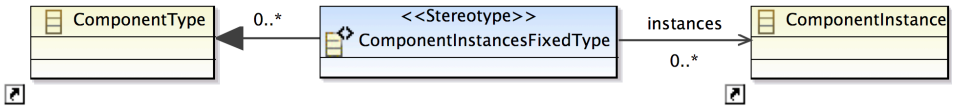


Figure 6.14: The *Fixed Component Instantiation* profile of the IAL

Formalization Definition 89 gives a formal description of the profile.

Definition 89: Component Instantiation Fixed Profile

The Component Instantiation Fixed profile $P_{FixedComponentInstantiation}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}\}$$

$$Stereotypes := \{s_{CompInstFixTyp}\}$$

$$References := \{r_{CompInstFixTyp\_inst}\}$$

The elements are named as follows:

$$\begin{aligned} name(s_{CompInstFixTyp}) &= \text{ComponentInstancesFixedType}, \\ name(r_{CompInstFixTyp\_inst}) &= \text{instances} \end{aligned}$$

The stereotypes extend the following classes:

$$\text{ComponentInstancesFixedType} \xrightarrow{extends} \text{ComponentType}$$

The attributes and references are defined as follows:

$$\text{ComponentInstancesFixedType.instances} \xrightarrow{isOfType} \text{ComponentInstance}$$

Per Session Component Instantiation

In the Per Session Component Instantiation profile (see Figure 6.15) a new component instance is created for each user session. Instantiations per user session is known from architecture implementation languages from the domain of information systems, where server-side user sessions are handled. JEE e.g. provides stateful session beans for this case. In architecture specification languages, this type of component instantiation is uncommon. To model this instantiation type, the stereotype *ComponentInstancePerSessionType* has to be applied to a component type.

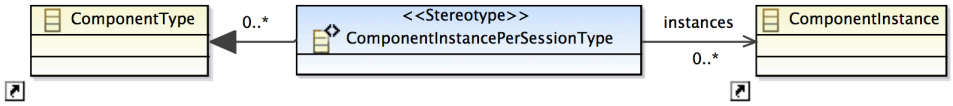


Figure 6.15: The *Per Session Component Instantiation* profile of the IAL

Formalization Definition 90 gives a formal description of the profile.

Definition 90: Component Instantiation Persession Profile

The Component Instantiation Persession profile $P_{PerSessionComponentInstantiation}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}\}$$

$$Stereotypes := \{s_{CompInstPerSessionTyp}\}$$

$$References := \{r_{CompInstPerSessionTyp\_inst}\}$$

The elements are named as follows:

$$\begin{aligned} name(s_{CompInstPerSessionTyp}) &= \text{ComponentInstancePerSessionType}, \\ name(r_{CompInstPerSessionTyp\_inst}) &= \text{instances} \end{aligned}$$

The stereotypes extend the following classes:

$$\text{ComponentInstancePerSessionType} \xrightarrow{extends} \text{ComponentType}$$

The attributes and references are defined as follows:

$$\text{ComponentInstancePerSessionType.instances} \xrightarrow{isOfType} \text{ComponentInstance}$$

Pooled Component Instantiation

The Pooled Component Instantiation profile (see Figure 6.16) can be used to model an instantiation behaviour, where a pool of instances should be available. This instantiation type is often used in architecture implementation languages for creating access to components with a high performance. E.g. in EJB, stateless session beans can be used to create this behaviour.

To model such an instantiation behaviour, the stereotype *ComponentInstancePooledType* can be applied to a component type. The minimal and maximal number of instances can be given. The referenced component instance is a representative of the pool elements at design time. The algorithm to choose the number of instances is represented by the *PoolingStrategy*. Its implementation is subject to the execution runtime.

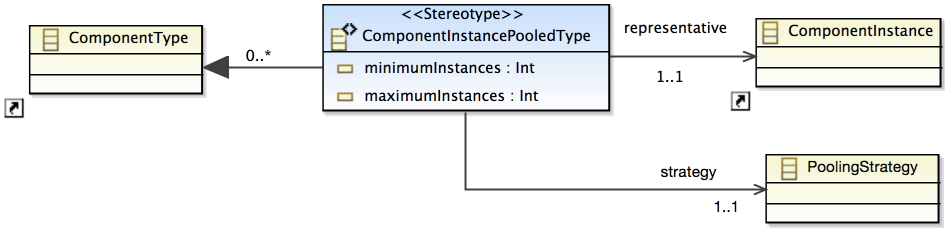


Figure 6.16: The *Pooled Component Instantiation* profile of the IAL

Formalization Definition 91 gives a formal description of the profile.

Definition 91: Component Instantiation Pooled Profile

The Component Instantiation Pooled profile $P_{PooledComponentInstantiation}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}\}$$

$$Stereotypes := \{s_{CompInstPoolTyp}\}$$

$$Classes := \{c_{PoolSt}\}$$

$$Attributes := \{a_{CompInstPoolTyp\_minimumInst}, a_{CompInstPoolTyp\_maxInst}\}$$

$$References := \{r_{CompInstPoolTyp\_rep}, r_{CompInstPoolTyp\_st}\}$$

$$Containments := \{r_{CompInstPoolTyp\_st}\}$$

The elements are named as follows:

$$\begin{aligned} name(s_{CompInstPoolTyp}) &= \text{ComponentInstancePooledType}, \\ name(c_{PoolSt}) &= \text{PoolingStrategy}, \\ name(a_{CompInstPoolTyp\_minimumInst}) &= \text{minimumInstances}, \\ name(a_{CompInstPoolTyp\_maxInst}) &= \text{maximumInstances}, \\ name(r_{CompInstPoolTyp\_rep}) &= \text{representative}, \\ name(r_{CompInstPoolTyp\_st}) &= \text{strategy} \end{aligned}$$

The stereotypes extend the following classes:

$$\text{ComponentInstancePooledType} \xrightarrow{extends} \text{ComponentType}$$

The attributes and references are defined as follows:

$$\begin{aligned} \text{ComponentInstancePooledType.minimumInstances} &\xrightarrow{isOfType} \text{Int}, \\ \text{ComponentInstancePooledType.maximumInstances} &\xrightarrow{isOfType} \text{Int}, \\ \text{ComponentInstancePooledType.representative} &\xrightarrow{cardinality} \text{1..1}, \\ \text{ComponentInstancePooledType.representative} &\xrightarrow{isOfType} \text{ComponentInstance}, \\ \text{ComponentInstancePooledType.strategy} &\xrightarrow{cardinality} \text{1..1}, \\ \text{ComponentInstancePooledType.strategy} &\xrightarrow{isOfType} \text{PoolingStrategy} \end{aligned}$$

6.5.5 Component State

The component state profiles model whether a component is stateful or stateless. While it is not allowed to declare multiple of these stereotypes on the same component type, these profiles can be used simultaneously in an architecture. Stateful components are e.g. used in JEE architectures. Many languages do not explicitly distinguish between stateful and stateless components. But some languages declare e.g. state machines as behaviour, which implies a statefulness.

Stateful Components

The Stateful Components profile (see Figure 6.17) can declare stateful components. A stateful component has a state, and its behaviour changes based on the state. To denote a component type stateful the stereotype *StatefulComponentType* can be applied to it.

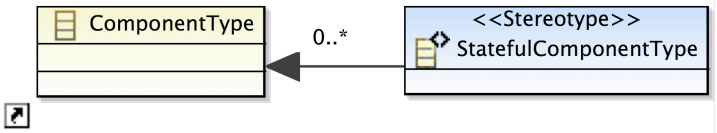


Figure 6.17: The *Stateful Components* profile of the IAL

Formalization Definition 92 gives a formal description of the profile.

Definition 92: Component State Stateful Profile

The Component State Stateful profile $P_{StatefulComponents}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}\}$$

$$Stereotypes := \{s_{StCompTyp}\}$$

The elements are named as follows:

$$name(s_{StCompTyp}) = \text{StatefulComponentType}$$

The stereotypes extend the following classes:

$$\text{StatefulComponentType} \xrightarrow{extends} \text{ComponentType}$$

Stateless Components

The Stateless Components profile (see Figure 6.18) can declare stateless components. A stateless component does not change its behaviour based on a state. It may however technically have a state that does not change the functionality. E.g. it can have a cache of data that is filled over time to increase the performance. To denote a component type stateful the stereotype *StatelessComponentType* can be applied to it.

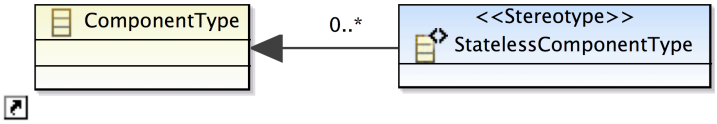


Figure 6.18: The *Stateless Components* profile of the IAL

Formalization Definition 93 gives a formal description of the profile.

Definition 93: Component State Stateless Profile

The Component State Stateless profile $P_{StatelessComponents}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}\}$$

$$Stereotypes := \{s_{StCompTyp}\}$$

The elements are named as follows:

$$name(s_{StCompTyp}) = \text{StatelessComponentType}$$

The stereotypes extend the following classes:

$$\text{StatelessComponentType} \xrightarrow{extends} \text{ComponentType}$$

6.5.6 Behaviour

Behaviour profiles can be used to give component types a formal behaviour description. One behaviour profile has been created in this thesis, to give an example how such profiles work and integrate with the structural profiles.

State Machine

State machines are formal descriptions of behaviour, based on states and transitions between states. The State Machine profile (see Figure 6.19) implements state machines based on the description of Peled et al. in [PGS01], without channels.

The stereotype *StateMachineBasedComponentType* can be applied to a component type. Such a component type has a *StateMachine* object that represents the component's state machine. The *StateMachine* class has named *States*, one of which is the initial state. States have named *Transitions* which can be triggered. The triggering of a transition is guarded by a *Guard*, which evaluates based on the context, whether a transition may be triggered. Triggering an execution leads to another state, the *targetState*. The *StateMachine* class has an actor, which acts as an interface between the state machine and the context. The actor can be used by a *Guard* before the execution of *Transitions* to validate that a transition can be triggered. During the execution of a transition, the actor can be used to influence the context.

The use of this profile does not exclude the use of the profile Stateless Components. It would be a contradiction to use the stereotype *StatelessComponentType* on a component type which has a state machine applied, but other component types might still be stateless. It is not strictly necessary to apply the *StatefulComponentType* stereotype to component types with state machines.

This profile for state machines is based on the Balz's ideas (see Section 3.2.4). In contrast to that related work, the state machines in this profile do not explicitly declare variables. Instead, the actor may include variables that can be set.

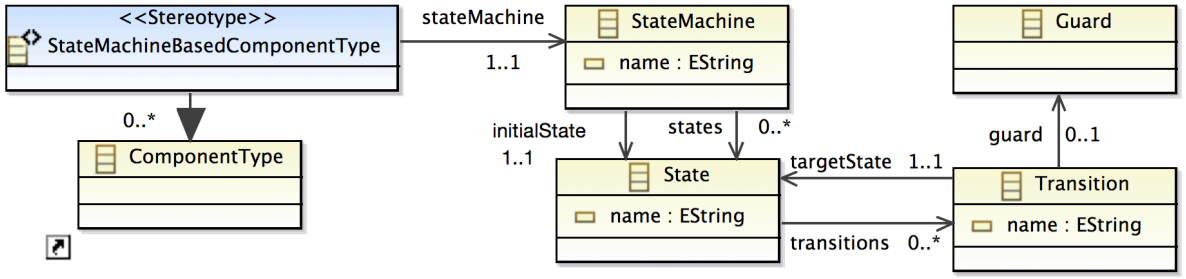


Figure 6.19: The *State Machine* profile of the IAL

Formalization Definition 94 gives a formal description of the profile.

Definition 94: State Machine Profile

The State Machine profile $P_{StateMachine}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}\}$$

$$Stereotypes := \{s_{StMacBasedCompTyp}\}$$

$$Classes := \{c_{StMac}, c_{St}, c_{Tr}, c_{Guar}\}$$

$$Attributes := \{a_{StMac\_name}, a_{St\_name}, a_{Tr\_name}\}$$

$$References := \{r_{StMacBasedCompTyp\_stMac}, r_{StMac\_st}, r_{St\_tr}, r_{Tr\_tarSt}, r_{Tr\_guar}, r_{StMac\_initSt}\}$$

$$Containments := \{r_{StMacBasedCompTyp\_stMac}, r_{StMac\_st}, r_{St\_tr}, r_{Tr\_guar}\}$$

The elements are named as follows:

$$\begin{aligned}
name(s_{StMacBasedCompTyp}) &= \text{StateMachineBasedComponentType},\\
name(c_{StMac}) &= \text{StateMachine},\\
name(c_{St}) &= \text{State},\\
name(c_{Tr}) &= \text{Transition},\\
name(c_{Guar}) &= \text{Guard},\\
name(a_{StMac_name}) &= \text{name},\\
name(a_{St_name}) &= \text{name},\\
name(a_{Tr_name}) &= \text{name},\\
name(r_{StMacBasedCompTyp_stMac}) &= \text{stateMachine},\\
name(r_{StMac_st}) &= \text{states},\\
name(r_{St_tr}) &= \text{transitions},\\
name(r_{Tr_tarSt}) &= \text{targetState},
\end{aligned}$$

$$name(r_{Tr\_guar}) = \text{guard},$$
$$name(r_{StMac\_initSt}) = \text{initialState}$$

The stereotypes extend the following classes:

$$\text{StateMachineBasedComponentType} \xrightarrow{extends} \text{ComponentType}$$

The attributes and references are defined as follows:

$$\text{StateMachine.name} \xrightarrow{isOfType} \text{String},$$
$$\text{State.name} \xrightarrow{isOfType} \text{String},$$
$$\text{Transition.name} \xrightarrow{isOfType} \text{String},$$
$$\text{StateMachineBasedComponentType.stateMachine} \xrightarrow{cardinality} \text{1..1},$$
$$\text{StateMachineBasedComponentType.stateMachine} \xrightarrow{isOfType} \text{StateMachine},$$
$$\text{StateMachine.states} \xrightarrow{isOfType} \text{State},$$
$$\text{State.transitions} \xrightarrow{isOfType} \text{Transition},$$
$$\text{Transition.targetState} \xrightarrow{cardinality} \text{1..1},$$
$$\text{Transition.targetState} \xrightarrow{isOfType} \text{State},$$
$$\text{Transition.guard} \xrightarrow{cardinality} \text{0..1},$$
$$\text{Transition.guard} \xrightarrow{isOfType} \text{Guard},$$
$$\text{StateMachine.initialState} \xrightarrow{cardinality} \text{1..1},$$
$$\text{StateMachine.initialState} \xrightarrow{isOfType} \text{State}$$

6.5.7 Connectors

The connectors profiles handle connections between components, based on the type of interfaces.

Connector

Connectors connect provisions with requirements, or delegate requirements or provisions. Because not all architecture languages use explicit connectors, the connectors are modelled as a profile. The Connector profile (see Figure 6.20) declares the existence of a connector type, connectors, and connector instances. *ConnectorTypes* are distinguished by names. They imply protocols and connection properties. An example is a Web Service connector. The semantics of the connector type are not modelled in this profile. *Connectors* can be used to connect interface requirements to provisions, while *ConnectorInstances* can be used to connect the corresponding requirement instances to provision instance. Connectors and their instances are abstract in the context of this profile and are refined using subsequent profiles. The specializing profiles for connectors can be used simultaneously within an architecture. It is not allowed to apply multiple specialization stereotypes to the same connector. All specializing connector profiles in

this section require the common connector profile, because they reference its classes.

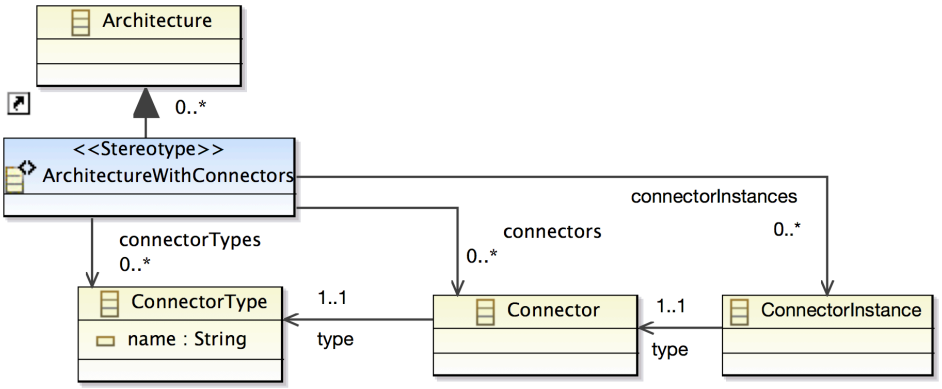


Figure 6.20: The *Connector* profile of the IAL

Formalization Definition 95 gives a formal description of the profile.

Definition 95: Connector Profile

The Connector profile $P_{Connector}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}\}$$

$$Stereotypes := \{s_{ArWitConnec}\}$$

$$Classes := \{c_{ConnecTyp}, c_{Connec}, c_{ConnecInst}\}$$

$$Attributes := \{a_{ConnecTyp\_name}\}$$

$$\begin{aligned} References := \{&r_{Connec\_typ},\\ &r_{ArWitConnec\_connecTyp},\\ &r_{ArWitConnec\_connec},\\ &r_{ArWitConnec\_connecInst},\\ &r_{ConnecInst\_typ}\} \end{aligned}$$

$$Containments := \{r_{ArWitConnec\_connecTyp}, r_{ArWitConnec\_connec}, r_{ArWitConnec\_connecInst}\}$$

The elements are named as follows:

$$\begin{aligned} name(s_{ArWitConnec}) &= \text{ArchitectureWithConnectors},\\ name(c_{ConnecTyp}) &= \text{ConnectorType},\\ name(c_{Connec}) &= \text{Connector},\\ name(c_{ConnecInst}) &= \text{ConnectorInstance}, \end{aligned}$$

$$
\begin{aligned}
name(a_{ConnecTyp\_name}) &= \text{name},\\
name(r_{Connec\_typ}) &= \text{type},\\
name(r_{ArWitConnec\_connecTyp}) &= \text{connectorTypes},\\
name(r_{ArWitConnec\_connec}) &= \text{connectors},\\
name(r_{ArWitConnec\_connecInst}) &= \text{connectorInstances},\\
name(r_{ConnecInst\_typ}) &= \text{type}
\end{aligned}
$$

The stereotypes extend the following classes:

$$\text{ArchitectureWithConnectors} \xrightarrow{extends} \text{Architecture}$$

The attributes and references are defined as follows:

$$
\begin{aligned}
\text{ConnectorType.name} &\xrightarrow{isOfType} \text{String},\\
\text{Connector.type} &\xrightarrow{cardinality} \text{1..1},\\
\text{Connector.type} &\xrightarrow{isOfType} \text{ConnectorType},\\
\text{ArchitectureWithConnectors.connectorTypes} &\xrightarrow{isOfType} \text{ConnectorType},\\
\text{ArchitectureWithConnectors.connectors} &\xrightarrow{isOfType} \text{Connector},\\
\text{ArchitectureWithConnectors.connectorInstances} &\xrightarrow{isOfType} \text{ConnectorInstance},\\
\text{ConnectorInstance.type} &\xrightarrow{cardinality} \text{1..1},\\
\text{ConnectorInstance.type} &\xrightarrow{isOfType} \text{Connector}
\end{aligned}
$$

Operation Call Connector

The Operation Call Connector profile (see Figure 6.21) is used to declare operation call interconnections. The stereotype *OperationCallConnector* can be applied to a *Connector*. It interconnects exactly one interface requirement with exactly one interface provision. The stereotype *OperationCallConnectorInstance* can be applied to a *ConnectorInstance*. It interconnects exactly one interface requirement instance with exactly one interface provision instance.

The profile requires the interconnected requirements and provisions and their instances to be based on operation interfaces as they are defined in Definition 82.

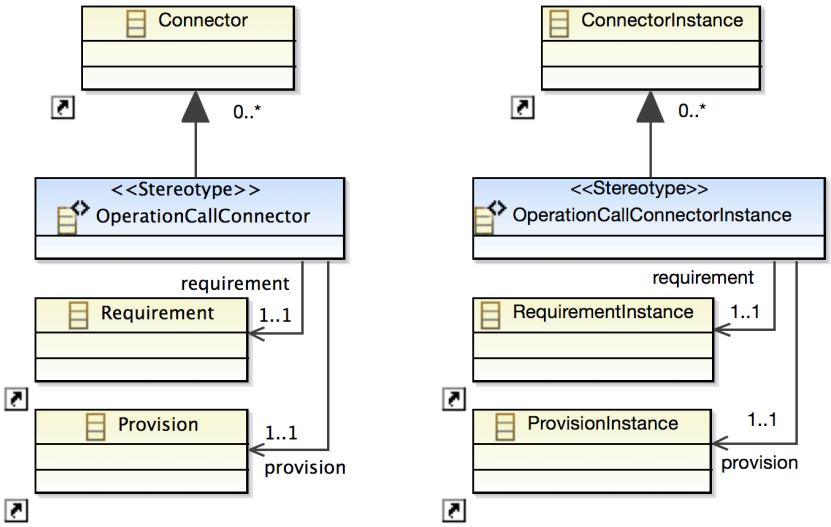


Figure 6.21: The *Operation Call Connector* profile of the IAL

Formalization Definition 96 gives a formal description of the profile.

Definition 96: Connector Operation Call Profile

The Connector Operation Call profile $P_{OperationCallConnector}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{P_{Connector}, M_{Meta}^{Kernel}\}$$
$$Stereotypes := \{s_{OpCalConnec}, s_{OpCalConnecInst}\}$$
$$References := \{r_{OpCalConnec\_req}, r_{OpCalConnec\_pr}, r_{OpCalConnecInst\_req}, r_{OpCalConnecInst\_pr}\}$$

The elements are named as follows:

$$\begin{aligned} name(s_{OpCalConnec}) &= \text{OperationCallConnector}, \\ name(s_{OpCalConnecInst}) &= \text{OperationCallConnectorInstance}, \\ name(r_{OpCalConnec\_req}) &= \text{requirement}, \\ name(r_{OpCalConnec\_pr}) &= \text{provision}, \\ name(r_{OpCalConnecInst\_req}) &= \text{requirement}, \\ name(r_{OpCalConnecInst\_pr}) &= \text{provision} \end{aligned}$$

The stereotypes extend the following classes:

$$\text{OperationCallConnector} \xrightarrow{extends} \text{Connector},$$
$$\text{OperationCallConnectorInstance} \xrightarrow{extends} \text{ConnectorInstance}$$

The attributes and references are defined as follows:

$$\text{OperationCallConnector.requirement} \xrightarrow{cardinality} \text{1..1},$$
$$\text{OperationCallConnector.requirement} \xrightarrow{isOfType} \text{Requirement},$$
$$\text{OperationCallConnector.provision} \xrightarrow{cardinality} \text{1..1},$$
$$\text{OperationCallConnector.provision} \xrightarrow{isOfType} \text{Provision},$$
$$\text{OperationCallConnectorInstance.requirement} \xrightarrow{cardinality} \text{1..1},$$
$$\text{OperationCallConnectorInstance.requirement} \xrightarrow{isOfType} \text{RequirementInstance},$$
$$\text{OperationCallConnectorInstance.provision} \xrightarrow{cardinality} \text{1..1},$$
$$\text{OperationCallConnectorInstance.provision} \xrightarrow{isOfType} \text{ProvisionInstance}$$

Event Dispatcher Connector

The Event Dispatcher Connector profile (see Figure 6.22) is used to declare event-based component interconnections. The stereotype *EventDispatcherConnector* can be applied to a *Connector*. It interconnects an arbitrary number of interface requirements with an arbitrary number of interface provisions. The stereotype *EventDispatcherConnectorInstance* can be applied to a *ConnectorInstance*. It interconnects an arbitrary number of interface requirement instances with an arbitrary number of interface provision instances.

The profile requires the interconnected requirements and provisions and their instances to be used on event-based interfaces as they are defined in Definition 83.

Formalization Definition 97 gives a formal description of the profile.

Definition 97: Connector Events Profile

The Connector Events profile $P_{EventDispatcherConnector}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{P_{Connector}, M_{Meta}^{Kernel}\}$$
$$Stereotypes := \{s_{EvDispConnec}, s_{EvDispConnecInst}\}$$
$$References := \{r_{EvDispConnec\_rec}, r_{EvDispConnec\_issuer}, r_{EvDispConnecInst\_issuer}, r_{EvDispConnecInst\_rec}\}$$

The elements are named as follows:

$$
\begin{aligned}
name(s_{EvDispConnec}) &= \text{EventDispatcherConnector},\\
name(s_{EvDispConnecInst}) &= \text{EventDispatcherConnectorInstance},\\
name(r_{EvDispConnec\_rec}) &= \text{receiver},\\
name(r_{EvDispConnec\_issuer}) &= \text{issuer},\\
name(r_{EvDispConnecInst\_issuer}) &= \text{issuer},\\
name(r_{EvDispConnecInst\_rec}) &= \text{receiver}
\end{aligned}
$$

The stereotypes extend the following classes:

$$
\begin{aligned}
\text{EventDispatcherConnector} &\xrightarrow{extends} \text{Connector},\\
\text{EventDispatcherConnectorInstance} &\xrightarrow{extends} \text{ConnectorInstance}
\end{aligned}
$$

The attributes and references are defined as follows:

$$
\begin{aligned}
\text{EventDispatcherConnector.receiver} &\xrightarrow{isOfType} \text{Requirement},\\
\text{EventDispatcherConnector.issuer} &\xrightarrow{isOfType} \text{Provision},\\
\text{EventDispatcherConnectorInstance.issuer} &\xrightarrow{isOfType} \text{ProvisionInstance},\\
\text{EventDispatcherConnectorInstance.receiver} &\xrightarrow{isOfType} \text{RequirementInstance}
\end{aligned}
$$

Delegation Connector

The Delegation Connector profile (see Figure 6.23) is used to declare a delegation of provisions and requirements as well as their instances from a parent to a child. Delegation connectors connect interface requirements of a parent component type with interface requirements of its child

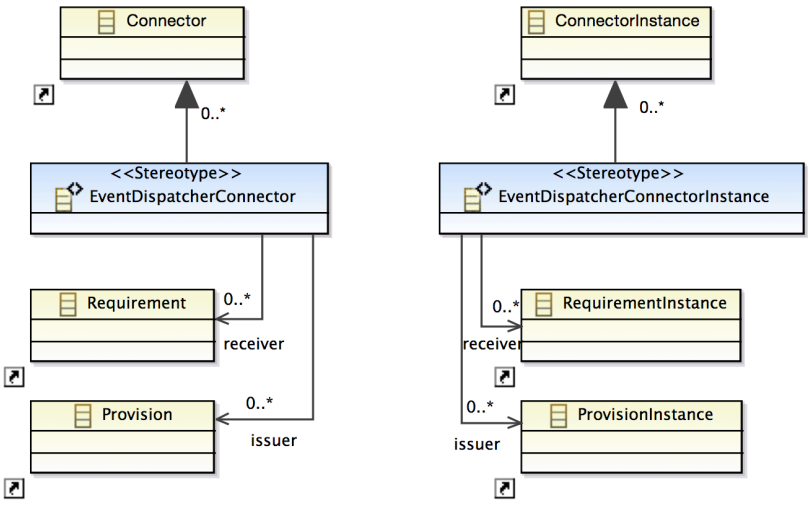


Figure 6.22: The *Event Dispatcher Connector* profile of the IAL

component type. To do so, the stereotype *RequirementDelegationConnector* can be applied to a *Connector*. The connector then declares the inner requirement that requires an interface, but it cannot be resolved within that scope, and the outer requirement, which is the delegatee and therefore gives the requirement to the higher level. Analogously the delegation of provisions is handled with the stereotype *ProvisionDelegationConnector*. The delegation of a provision means that incoming requests, events, etc. from the parent component are forwarded to a provision within the component type. For instances the stereotypes *RequirementDelegation-ConnectorInstance* and *ProvisionDelegationConnectorInstance* can be used.

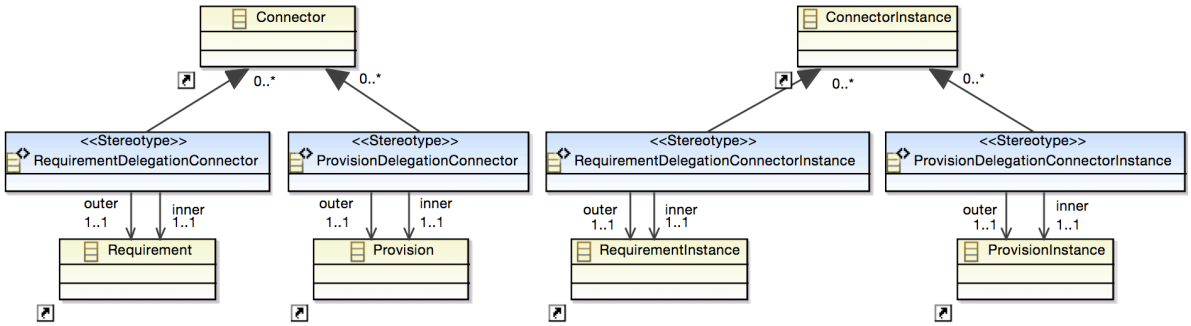


Figure 6.23: The *Delegation Connector* profile of the IAL

Formalization Definition 98 gives a formal description of the profile.

Definition 98: Delegation Connector Profile

The Connector Delegation profile $P_{DelegationConnector}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{P_{Connector}, M_{Meta}^{Kernel}\}$$

$$Stereotypes := \{s_{ReqDelConnec}, s_{PrDelConnec}, s_{ReqDelConnecInst}, s_{PrDelConnecInst}\}$$

$$\begin{aligned} References := \{&r_{ReqDelConnec\_out}, r_{ReqDelConnec\_inner}, r_{PrDelConnec\_out}, \\ &r_{PrDelConnec\_inner}, r_{ReqDelConnecInst\_out}, r_{ReqDelConnecInst\_inner}, \\ &r_{PrDelConnecInst\_out}, r_{PrDelConnecInst\_inner}\} \end{aligned}$$

The elements are named as follows:

$$\begin{aligned} name(s_{ReqDelConnec}) &= \text{RequirementDelegationConnector}, \\ name(s_{PrDelConnec}) &= \text{ProvisionDelegationConnector}, \\ name(s_{ReqDelConnecInst}) &= \text{RequirementDelegationConnectorInstance}, \\ name(s_{PrDelConnecInst}) &= \text{ProvisionDelegationConnectorInstance}, \end{aligned}$$

$$
\begin{aligned}
name(r_{ReqDelConnec\_out}) &= \text{outer},\\
name(r_{ReqDelConnec\_inner}) &= \text{inner},\\
name(r_{PrDelConnec\_out}) &= \text{outer},\\
name(r_{PrDelConnec\_inner}) &= \text{inner},\\
name(r_{ReqDelConnecInst\_out}) &= \text{outer},\\
name(r_{ReqDelConnecInst\_inner}) &= \text{inner},\\
name(r_{PrDelConnecInst\_out}) &= \text{outer},\\
name(r_{PrDelConnecInst\_inner}) &= \text{inner}
\end{aligned}
$$

The stereotypes extend the following classes:

$$
\begin{aligned}
\text{RequirementDelegationConnector} &\xrightarrow{extends} \text{Connector},\\
\text{ProvisionDelegationConnector} &\xrightarrow{extends} \text{Connector},\\
\text{RequirementDelegationConnectorInstance} &\xrightarrow{extends} \text{ConnectorInstance},\\
\text{ProvisionDelegationConnectorInstance} &\xrightarrow{extends} \text{ConnectorInstance}
\end{aligned}
$$

The attributes and references are defined as follows:

$$
\begin{aligned}
\text{RequirementDelegationConnector.outer} &\xrightarrow{cardinality} \text{1..1},\\
\text{RequirementDelegationConnector.outer} &\xrightarrow{isOfType} \text{Requirement},\\
\text{RequirementDelegationConnector.inner} &\xrightarrow{cardinality} \text{1..1},\\
\text{RequirementDelegationConnector.inner} &\xrightarrow{isOfType} \text{Requirement},\\
\text{ProvisionDelegationConnector.outer} &\xrightarrow{cardinality} \text{1..1},\\
\text{ProvisionDelegationConnector.outer} &\xrightarrow{isOfType} \text{Provision},\\
\text{ProvisionDelegationConnector.inner} &\xrightarrow{cardinality} \text{1..1},\\
\text{ProvisionDelegationConnector.inner} &\xrightarrow{isOfType} \text{Provision},\\
\text{RequirementDelegationConnectorInstance.outer} &\xrightarrow{cardinality} \text{1..1},\\
\text{RequirementDelegationConnectorInstance.outer} &\xrightarrow{isOfType} \text{RequirementInstance},\\
\text{RequirementDelegationConnectorInstance.inner} &\xrightarrow{cardinality} \text{1..1},\\
\text{RequirementDelegationConnectorInstance.inner} &\xrightarrow{isOfType} \text{RequirementInstance},\\
\text{ProvisionDelegationConnectorInstance.outer} &\xrightarrow{cardinality} \text{1..1},\\
\text{ProvisionDelegationConnectorInstance.outer} &\xrightarrow{isOfType} \text{ProvisionInstance},\\
\text{ProvisionDelegationConnectorInstance.inner} &\xrightarrow{cardinality} \text{1..1},\\
\text{ProvisionDelegationConnectorInstance.inner} &\xrightarrow{isOfType} \text{ProvisionInstance}
\end{aligned}
$$

6.5.8 Datatypes

The datatypes profiles handle the existence of data types and their integration with other profiles.

Datatypes Common

The Datatypes Common profile (see Figure 6.24) is used to describe data types within the architecture. Data types are named elements that can be used to type e.g. operation parameters. They can have sub- and super types and operations with parameters.

The stereotype *ArchitectureWithCommonDataTypes* can be applied to an architecture to model that it declares data types. Data types have named *DataTypeOperations* that can be executed on the instances of data types at run time. These operations can have either a primitive type or a data type in the terms of this profile as return type. If a data type is declared as return type in terms of this profile, a value of the primitive type is ignored. Operations also have named *DataTypeOperationParameters*. These parameters are also of either a primitive type or a data type as type. Data types may have super types, which means that the subtypes inherit its operations. All profiles that specialize the Datatypes Common profile require this profile. Data types are common in architecture languages. They exist e.g. in JEE as entity beans, in PCM as data types.

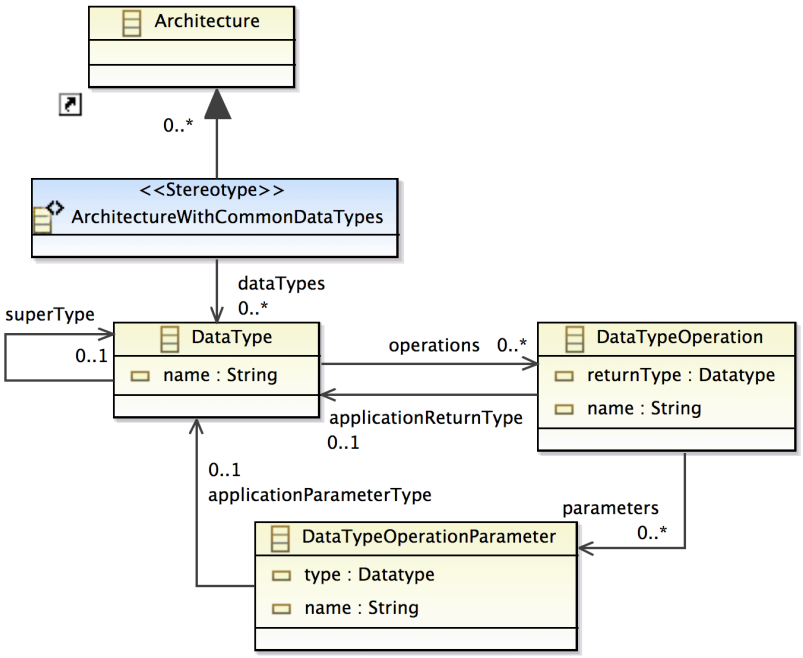


Figure 6.24: The *Datatypes Common* profile of the IAL

Formalization Definition 99 gives a formal description of the profile.

Definition 99: Datatype Common Profile

The Datatype Common profile $P_{DatatypesCommon}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}\}$$

$$Stereotypes := \{s_{ArWitCommonDatTyp}\}$$

$$Classes := \{c_{DatTyp}, c_{DatTypOp}, c_{DatTypOpPar}\}$$

$$Attributes := \{a_{DatTyp\_name}, a_{DatTypOp\_retTyp}, a_{DatTypOp\_name}, a_{DatTypOpPar\_typ}, a_{DatTypOpPar\_name}\}$$

$$References := \{r_{ArWitCommonDatTyp\_datTyp}, r_{DatTyp\_op}, r_{DatTypOp\_par}, r_{DatTypOpPar\_apParTyp}, r_{DatTypOp\_apRetTyp}, r_{DatTyp\_supTyp}\}$$

$$Containments := \{r_{ArWitCommonDatTyp\_datTyp}, r_{DatTyp\_op}, r_{DatTypOp\_par}\}$$

The elements are named as follows:

$$\begin{aligned}
name(s_{ArWitCommonDatTyp}) &= \text{ArchitectureWithCommonDataTypes},\\
name(c_{DatTyp}) &= \text{DataType},\\
name(c_{DatTypOp}) &= \text{DataTypeOperation},\\
name(c_{DatTypOpPar}) &= \text{DataTypeOperationParameter},\\
name(a_{DatTyp_name}) &= \text{name},\\
name(a_{DatTypOp_retTyp}) &= \text{returnType},\\
name(a_{DatTypOp_name}) &= \text{name},\\
name(a_{DatTypOpPar_typ}) &= \text{type},\\
name(a_{DatTypOpPar_name}) &= \text{name},\\
name(r_{ArWitCommonDatTyp_datTyp}) &= \text{dataTypes},\\
name(r_{DatTyp_op}) &= \text{operations},\\
name(r_{DatTypOp_par}) &= \text{parameters},\\
name(r_{DatTypOpPar_apParTyp}) &= \text{applicationParameterType},\\
name(r_{DatTypOp_apRetTyp}) &= \text{applicationReturnType},\\
name(r_{DatTyp_supTyp}) &= \text{superType}
\end{aligned}$$

The stereotypes extend the following classes:

$$\text{ArchitectureWithCommonDataTypes} \xrightarrow{extends} \text{Architecture}$$

The attributes and references are defined as follows:

$$\text{DataType.name} \xrightarrow{isOfType} \text{String},$$

$$\begin{aligned}
\text{DataTypeOperation.returnType} &\xrightarrow{isOfType} \text{Datatype},\\
\text{DataTypeOperation.name} &\xrightarrow{isOfType} \text{String},\\
\text{DataTypeOperationParameter.type} &\xrightarrow{isOfType} \text{Datatype},\\
\text{DataTypeOperationParameter.name} &\xrightarrow{isOfType} \text{String},\\
\text{ArchitectureWithCommonDataTypes.}&\\
\text{dataTypes} &\xrightarrow{isOfType} \text{DataType},\\
\text{DataType.operations} &\xrightarrow{isOfType} \text{DataTypeOperation},\\
\text{DataTypeOperation.parameters} &\xrightarrow{isOfType} \text{DataTypeOperationParameter},\\
\text{DataTypeOperationParameter.}&\\
\text{applicationParameterType} &\xrightarrow{cardinality} \text{0..1},\\
\text{DataTypeOperationParameter.}&\\
\text{applicationParameterType} &\xrightarrow{isOfType} \text{DataType},\\
\text{DataTypeOperation.applicationReturnType} &\xrightarrow{cardinality} \text{0..1},\\
\text{DataTypeOperation.applicationReturnType} &\xrightarrow{isOfType} \text{DataType},\\
\text{DataType.superType} &\xrightarrow{cardinality} \text{0..1},\\
\text{DataType.superType} &\xrightarrow{isOfType} \text{DataType}
\end{aligned}$$

Datatypes Operations

The Datatypes Operations profile (see Figure 6.25) allows to use the data types that were introduced in the Datatypes Common profile as return values of operations from the Operation Interfaces profile (see Definition 82), and as parameter types thereof. When the stereotype *OperationWithDataType* is applied, a data type must be set as return type, and the primitive type of the Operation Interfaces profile is ignored. The stereotype *OperationParameterDataType* works analogously for operation parameters. This profile requires the use of the Operation Interfaces profile, so that its stereotype can be applied to an operation interface.

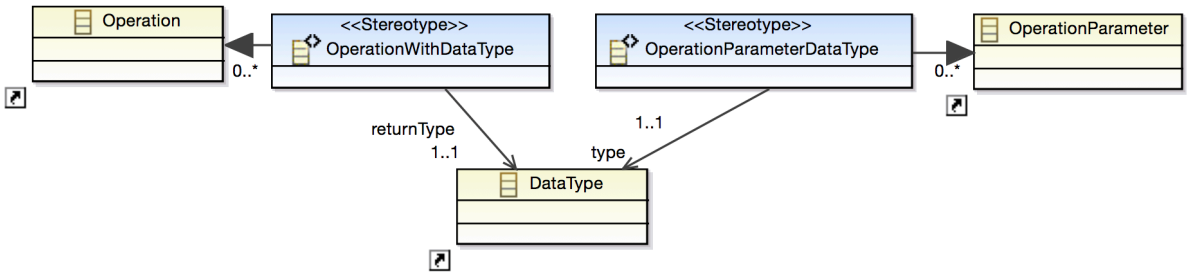


Figure 6.25: The *Datatypes Operations* profile of the IAL

Formalization Definition 100 gives a formal description of the profile.

Definition 100: Datatype Operations Profile

The Datatype Operations profile $P_{DatatypesOperations}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{P_{OperationInterfaces}, P_{DatatypesCommon}\}$$
$$Stereotypes := \{s_{OpWitDatTyp}, s_{OpParDatTyp}\}$$
$$References := \{r_{OpWitDatTyp\_retTyp}, r_{OpParDatTyp\_typ}\}$$

The elements are named as follows:

$$\begin{aligned} name(s_{OpWitDatTyp}) &= \text{OperationWithDataType}, \\ name(s_{OpParDatTyp}) &= \text{OperationParameterDataType}, \\ name(r_{OpWitDatTyp\_retTyp}) &= \text{returnType}, \\ name(r_{OpParDatTyp\_typ}) &= \text{type} \end{aligned}$$

The stereotypes extend the following classes:

$$\begin{aligned} \text{OperationWithDataType} &\xrightarrow{extends} \text{Operation}, \\ \text{OperationParameterDataType} &\xrightarrow{extends} \text{OperationParameter} \end{aligned}$$

The attributes and references are defined as follows:

$$\begin{aligned} \text{OperationWithDataType.returnType} &\xrightarrow{cardinality} \text{1..1}, \\ \text{OperationWithDataType.returnType} &\xrightarrow{isOfType} \text{DataType}, \\ \text{OperationParameterDataType.type} &\xrightarrow{cardinality} \text{1..1}, \\ \text{OperationParameterDataType.type} &\xrightarrow{isOfType} \text{DataType} \end{aligned}$$

Datatypes Events

Analogously to the Datatypes Operations profile for operations, the Datatypes Events profile (see Figure 6.26) allows to use the data types that were introduced in the Datatypes Common profile as parameters of events from the Event Interfaces profile (see Definition 83). When the stereotype *EventParameterDataType* is applied, a data type must be set as a type, and the primitive type of the profile Event Interfaces is ignored. This profile requires the use of the Event Interfaces profile, so that its stereotype can be applied to an event interface.

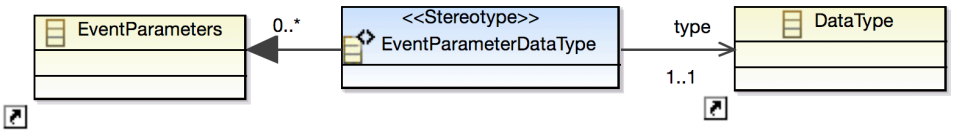


Figure 6.26: The *Datatypes Events* profile of the IAL

Formalization Definition 101 gives a formal description of the profile.

Definition 101: Datatype Events Profile

The Datatype Events profile $P_{DatatypesEvents}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{P_{EventInterfaces}, P_{DatatypesCommon}\}$$
$$Stereotypes := \{s_{EvParDatTyp}\}$$
$$References := \{r_{EvParDatTyp\_typ}\}$$

The elements are named as follows:

$$name(s_{EvParDatTyp}) = \text{EventParameterDataType},$$
$$name(r_{EvParDatTyp\_typ}) = \text{type}$$

The stereotypes extend the following classes:

$$\text{EventParameterDataType} \xrightarrow{extends} \text{EventParameters}$$

The attributes and references are defined as follows:

$$\text{EventParameterDataType.type} \xrightarrow{cardinality} \text{1..1},$$
$$\text{EventParameterDataType.type} \xrightarrow{isOfType} \text{DataType}$$

6.5.9 Deployment

The Deployment profile (see Figure 6.27) is used to describe the deployment of component instances and the packaging of component types into deployment fragments. Architecture implementation languages do not include deployment information, because the implementation can usually be run in multiple deployment configurations at run time. Specification languages use deployment information for analysis purposes. An example for deployment information in architecture specification languages is PCM. Deployment information related to the run time.

For describing deployment information, *ArchitectureWithDeploymentFragments* can be applied to an architecture. It owns deployment fragments, resource containers and allocation

contexts. *DeploymentFragments* are named elements that represent a deployable unit of an application, e.g. an executable, a library, or a JAR file. Deployment fragments are hierarchically organized and reference the component types that they contain.

ResourceContainers are named containers that can execute component instances. Examples for such containers are hosts or application servers. *AllocationContexts* map component instances to resource containers. The mapping means that a component instance is executed on the given resource container. For each component instance only one allocation context is allowed.

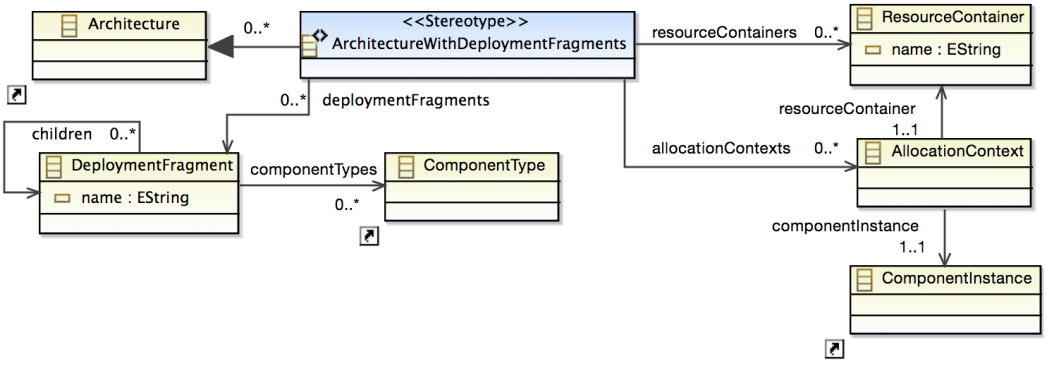


Figure 6.27: The *Deployment* profile of the IAL

Formalization Definition 102 gives a formal description of the profile.

Definition 102: Deployment Profile

The Deployment profile $P_{DatatypesEvents}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}\}$$

$$Stereotypes := \{s_{ArWitDepFr}\}$$

$$Classes := \{c_{DepFr}, c_{ResourCont}, c_{AlCont}\}$$

$$Attributes := \{a_{DepFr\_name}, a_{ResourCont\_name}\}$$

$$References := \{r_{ArWitDepFr\_depFr}, r_{DepFr\_chil}, r_{DepFr\_compTyp}, r_{ArWitDepFr\_resourCont}, r_{ArWitDepFr\_alCont}, r_{AlCont\_resourCont}, r_{AlCont\_compInst}\}$$

$$Containments := \{r_{ArWitDepFr\_depFr}, r_{DepFr\_chil}, r_{ArWitDepFr\_resourCont}, r_{ArWitDepFr\_alCont}\}$$

The elements are named as follows:

$$
\begin{aligned}
name(s_{ArWitDepFr}) &= \text{ArchitectureWithDeploymentFragments},\\
name(c_{DepFr}) &= \text{DeploymentFragment},\\
name(c_{ResourCont}) &= \text{ResourceContainer},\\
name(c_{AlCont}) &= \text{AllocationContext},\\
name(a_{DepFr\_name}) &= \text{name},\\
name(a_{ResourCont\_name}) &= \text{name},\\
name(r_{ArWitDepFr\_depFr}) &= \text{deploymentFragments},\\
name(r_{DepFr\_chil}) &= \text{children},\\
name(r_{DepFr\_compTyp}) &= \text{componentTypes},\\
name(r_{ArWitDepFr\_resourCont}) &= \text{resourceContainers},\\
name(r_{ArWitDepFr\_alCont}) &= \text{allocationContexts},\\
name(r_{AlCont\_resourCont}) &= \text{resourceContainer},\\
name(r_{AlCont\_compInst}) &= \text{componentInstance}
\end{aligned}
$$

The stereotypes extend the following classes:

$$\text{ArchitectureWithDeploymentFragments} \xrightarrow{extends} \text{Architecture}$$

The attributes and references are defined as follows:

$$\text{DeploymentFragment.name} \xrightarrow{isOfType} \text{String},$$

$$\text{ResourceContainer.name} \xrightarrow{isOfType} \text{String},$$

ArchitectureWithDeploymentFragments.

$$\text{deploymentFragments} \xrightarrow{isOfType} \text{DeploymentFragment},$$

$$\text{DeploymentFragment.children} \xrightarrow{isOfType} \text{DeploymentFragment},$$

$$\text{DeploymentFragment.componentTypes} \xrightarrow{isOfType} \text{ComponentType},$$

ArchitectureWithDeploymentFragments.

$$\text{resourceContainers} \xrightarrow{isOfType} \text{ResourceContainer},$$

ArchitectureWithDeploymentFragments.

$$\text{allocationContexts} \xrightarrow{isOfType} \text{AllocationContext},$$

$$\text{AllocationContext.resourceContainer} \xrightarrow{cardinality} \text{1..1},$$

$$\text{AllocationContext.resourceContainer} \xrightarrow{isOfType} \text{ResourceContainer},$$

$$\text{AllocationContext.componentInstance} \xrightarrow{cardinality} \text{1..1},$$

$$\text{AllocationContext.componentInstance} \xrightarrow{isOfType} \text{ComponentInstance}$$

6.5.10 Namespace

The Namespace profile (see Figure 6.28) introduces namespaces into the IAL. Namespaces are hierarchically organized named elements. Interfaces and component types can be related to namespaces to collect semantically coupled elements. This profile can be used to represent namespaces as they are e.g. used in Java applications. In Java these namespaces are called *packages*. The stereotype *ArchitectureWithNamespaces* is used to signal that components and interfaces are located within specific namespaces.

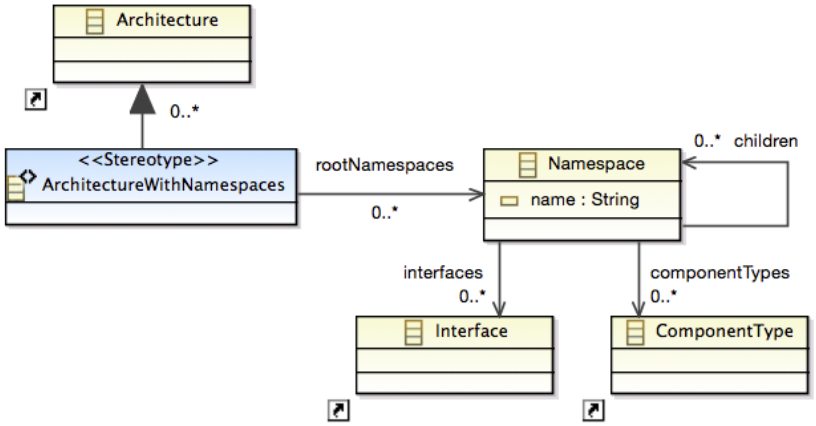


Figure 6.28: The *Namespace* profile of the IAL

Formalization Definition 103 gives a formal description of the profile.

Definition 103: Namespaces Profile

The Namespaces profile $P_{DatatypesEvents}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}\}$$

$$Stereotypes := \{s_{ArWitNamesp}\}$$

$$Classes := \{c_{Namesp}\}$$

$$Attributes := \{a_{Namesp\_name}\}$$

$$References := \{r_{ArWitNamesp\_rootNamesp}, r_{Namesp\_chil}, r_{Namesp\_int}, r_{Namesp\_compTyp}\}$$

$$Containments := \{r_{ArWitNamesp\_rootNamesp}, r_{Namesp\_chil}\}$$

The elements are named as follows:

$$\begin{aligned} name(s_{ArWitNamesp}) &= \text{ArchitectureWithNamespaces}, \\ name(c_{Namesp}) &= \text{Namespace}, \\ name(a_{Namesp\_name}) &= \text{name}, \end{aligned}$$

$$
\begin{aligned}
name(r_{ArWitNamesp\_rootNamesp}) &= \text{rootNamespaces},\\
name(r_{Namesp\_chil}) &= \text{children},\\
name(r_{Namesp\_int}) &= \text{interfaces},\\
name(r_{Namesp\_compTyp}) &= \text{componentTypes}
\end{aligned}
$$

The stereotypes extend the following classes:

$$\text{ArchitectureWithNamespaces} \xrightarrow{extends} \text{Architecture}$$

The attributes and references are defined as follows:

$$
\begin{aligned}
\text{Namespace.name} &\xrightarrow{isOfType} \text{String},\\
\text{ArchitectureWithNamespaces.rootNamespaces} &\xrightarrow{isOfType} \text{Namespace},\\
\text{Namespace.children} &\xrightarrow{isOfType} \text{Namespace},\\
\text{Namespace.interfaces} &\xrightarrow{isOfType} \text{Interface},\\
\text{Namespace.componentTypes} &\xrightarrow{isOfType} \text{ComponentType}
\end{aligned}
$$

6.5.11 Software Quality

The software quality profiles consider quality concerns of the architecture. As examples how quality profiles integrate with the IAL, two quality concerns are defined: performance in terms of time resource demand and security in terms of secure information flow properties.

Time Resource Demand

The Time Resource Demand profile (see Figure 6.29) describes the resource demand of operations in operation-based interfaces as they are defined in Definition 82. Time resource demands on operations are e.g. used in PCM, where static values, or probability functions can be stated for declaring resource demands. The stereotype *TimeResourceDemand* can be applied to operations. It takes as attribute a duration of operations. In this exemplary profile, no specific notation is defined. The duration can be given in milliseconds, seconds, or e.g. as probability function.

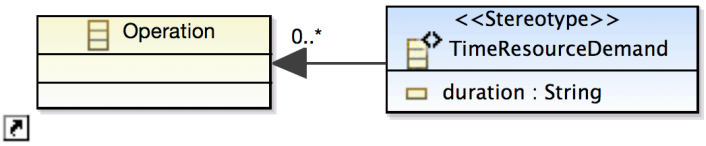


Figure 6.29: The *Time Resource Demand* profile of the IAL

Formalization Definition 104 gives a formal description of the profile.

Definition 104: Time Resource Demand Profile

The Quality Time profile $P_{TimeResourceDemand}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{P_{OperationInterfaces}\}$$

$$Stereotypes := \{s_{TimeResourDemand}\}$$

$$Attributes := \{a_{TimeResourDemand\_dur}\}$$

The elements are named as follows:

$$\begin{aligned} name(s_{TimeResourDemand}) &= \text{TimeResourceDemand}, \\ name(a_{TimeResourDemand\_dur}) &= \text{duration} \end{aligned}$$

The stereotypes extend the following classes:

$$\text{TimeResourceDemand} \xrightarrow{extends} \text{Operation}$$

The attributes and references are defined as follows:

$$\text{TimeResourceDemand.duration} \xrightarrow{isOfType} \text{String}$$

Secure Information Flow

The Secure Information Flow profile (see Figure 6.30) defines the information necessary to implement the secure information flow (SIF) concept [RJ12, Smi07]. With SIF annotations, software defines security levels, ordered by a partially ordered set (POSET). Executable units define a minimal security level under which they are allowed to be executed. An execution always happens in the context of a security level (security context). The execution of an executable unit is only allowed when the security context of the execution is the same as the declared security level of the executable unit, or higher in the POSET. This description can be the basis for a SIF analysis as described by Mantel [Man03].

In this profile, executable units are interfaces and operations. Existing security levels and the POSET are declared in component types and namespaces. Therefor the stereotype *DefSecurityLevels* can be applied to component types or namespaces. It contains a set of *SecurityLevelEntities* which represent named security levels. A set of *SecurityLevelPosets*, owned by the *DefSecurityLevels* stereotype, bring the security level entities in a partial order.

The stereotype *SecurityLevel* can be applied to interfaces and operations. It references *SecurityLevelEntities*, which are owned by the component type or namespace, that the interface or operation belongs to. The referenced security level entity represents the security context in which an interface or an operation is allowed to be used. The stereotype *SIFProperty* declares *Basic Security Predicates* (BSP) [Man03, Chapter 3]. BSPs are security predicates for secure information flows. An example is the property is *Strict Removal* (SR), which describes that it is required that no information about confidential events can be inferred from publicly visible events. The names of the upper and lower BSPs are entered as attributes. Tools can use this information for analysing secure information flow properties. This profile requires operation type interfaces (see Definition 82), and the Namespaces profile (see Definition 103).

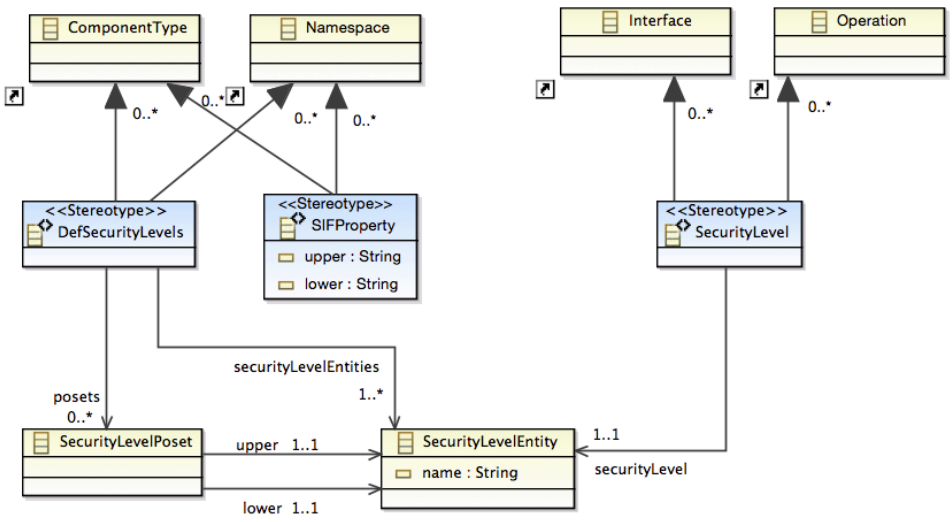


Figure 6.30: The *Secure Information Flow* profile of the IAL

Formalization Definition 105 gives a formal description of the profile.

Definition 105: Secure Information Flow Profile

The Secure Information Flow profile $P_{SecureInformationFlow}$ is defined as follows. Empty sets are not explicitly stated:

$$B := \{M_{Meta}^{Kernel}, P_{Namespace}, P_{OperationInterfaces}\}$$

$$Stereotypes := \{s_{DefSecLev}, s_{SecLev}, s_{SIFPr}\}$$

$$Classes := \{c_{SecLevPoset}, c_{SecLevEnt}\}$$

$$Attributes := \{a_{SecLevEnt\_name}, a_{SIFPr\_up}, a_{SIFPr\_low}\}$$

$$References := \{r_{DefSecLev\_poset}, r_{SecLevPoset\_up}, r_{SecLevPoset\_low}, r_{DefSecLev\_secLevEnt}, r_{SecLev\_secLev}\}$$

$$Containments := \{r_{DefSecLev\_poset}, r_{DefSecLev\_secLevEnt}\}$$

The elements are named as follows:

$$\begin{aligned} name(s_{DefSecLev}) &= \text{DefSecurityLevels}, \\ name(s_{SecLev}) &= \text{SecurityLevel}, \\ name(s_{SIFPr}) &= \text{SIFProperty}, \\ name(c_{SecLevPoset}) &= \text{SecurityLevelPoset}, \\ name(c_{SecLevEnt}) &= \text{SecurityLevelEntity}, \end{aligned}$$

$$
\begin{aligned}
name(a_{SecLevEnt\_name}) &= \text{name},\\
name(a_{SIFPr\_up}) &= \text{upper},\\
name(a_{SIFPr\_low}) &= \text{lower},\\
name(r_{DefSecLev\_poset}) &= \text{posets},\\
name(r_{SecLevPoset\_up}) &= \text{upper},\\
name(r_{SecLevPoset\_low}) &= \text{lower},\\
name(r_{DefSecLev\_secLevEnt}) &= \text{securityLevelEntities},\\
name(r_{SecLev\_secLev}) &= \text{securityLevel}
\end{aligned}
$$

The stereotypes extend the following classes:

$$
\begin{aligned}
\text{DefSecurityLevels} &\xrightarrow{extends} \text{ComponentType},\\
\text{DefSecurityLevels} &\xrightarrow{extends} \text{Namespace},\\
\text{SecurityLevel} &\xrightarrow{extends} \text{Interface},\\
\text{SecurityLevel} &\xrightarrow{extends} \text{Operation},\\
\text{SIFProperty} &\xrightarrow{extends} \text{Namespace},\\
\text{SIFProperty} &\xrightarrow{extends} \text{ComponentType}
\end{aligned}
$$

The attributes and references are defined as follows:

$$
\begin{aligned}
\text{SecurityLevelEntity.name} &\xrightarrow{isOfType} \text{String},\\
\text{SIFProperty.upper} &\xrightarrow{isOfType} \text{String},\\
\text{SIFProperty.lower} &\xrightarrow{isOfType} \text{String},\\
\text{DefSecurityLevels.posets} &\xrightarrow{isOfType} \text{SecurityLevelPoset},\\
\text{SecurityLevelPoset.upper} &\xrightarrow{cardinality} \text{1..1},\\
\text{SecurityLevelPoset.upper} &\xrightarrow{isOfType} \text{SecurityLevelEntity},\\
\text{SecurityLevelPoset.lower} &\xrightarrow{cardinality} \text{1..1},\\
\text{SecurityLevelPoset.lower} &\xrightarrow{isOfType} \text{SecurityLevelEntity},\\
\text{DefSecurityLevels.securityLevelEntities} &\xrightarrow{cardinality} \text{1..}*,\\
\text{DefSecurityLevels.securityLevelEntities} &\xrightarrow{isOfType} \text{SecurityLevelEntity},\\
\text{SecurityLevel.securityLevel} &\xrightarrow{cardinality} \text{1..1},\\
\text{SecurityLevel.securityLevel} &\xrightarrow{isOfType} \text{SecurityLevelEntity}
\end{aligned}
$$

This section presented a set of profiles for the IAL, for describing common architectural concerns. The IAL can be subject to evolution by creating new profiles, by extending the kernel, or by evolving existing profiles.

6.6 Evaluation Regarding the Requirements Towards a Transformation Model Language

As argued in Section 6.2.3, the selection of the profiles strategy and EMF Profiles as technology for the IAL lays the basis for fulfilling the requirements IL-R1, IL-R2, IL-R3, and IL-R5. The IAL, as it is described in this section, fulfills the requirements as follows:

IL-R1 It must be possible to extend the meta model with new first class entities.

The use of the profile strategy allows for adding arbitrary abstract syntax elements to a meta model. A profile can define new classes, which can be contained by a stereotype. The IAL can be extended by creating profiles with stereotypes that extend classes of the language kernel or of other profiles in the IAL. Therefore the language is prepared to integrate arbitrary architectural concepts.

IL-R2 It must be possible to extend existing elements of the meta model with new properties.

It is possible to extend meta model elements with new properties using the profile strategy. By declaring and applying stereotypes, new attributes and references can be added to existing meta model elements.

IL-R3 It must be possible to model multiple concerns regarding a meta model element simultaneously.

The profile strategy allows for modelling multiple concerns regarding a model element simultaneously, by extending a class with multiple stereotypes and applying multiple stereotypes to an object. It is also possible to model conflicting information—e.g. flat and scoped component type hierarchies—simultaneously. An interpreter decides, which information is to be used in the specific context.

IL-R4 It must be possible to represent components, interfaces, and connectors.

First class entities exist for components and abstract interfaces in the language kernel. Connectors are declared in the profile Connector, and specialized interfaces are declared in their own profiles.

IL-R5 The meta model must be easily usable with tools working with Ecore meta models.

Meta models, as they are declared in Section 5.4.1, can be represented using a subset of Ecore. Profiles as they are defined for the IAL can be represented using EMF Profiles. An implementation of the IAL exists using Ecore and EMF profiles (see Section 9.2). All Ecore tools are usable with the meta model and its models.

6.7 Summary

This chapter first discussed the requirements for an architecture translation model language in the Explicitly Integrated Architecture Process and strategies, that allow to fulfill these requirements. The Intermediate Architecture Description Language, an architecture translation model language that is built to satisfy these requirements, was presented. First, an overview of the language was given, including the interdependencies between profiles of the IAL. Then the language's kernel and the profiles were described in detail. At last the language was evaluated regarding the requirements.

The role if the IAL in the Explicitly Integrated Architecture approach is that of an intermediate language for transformations between architecture implementation and specification languages. The next chapter describes these transformations.

7 Architecture Model Transformations

This chapter describes the model transformations between architecture specification languages, the Intermediate Architecture Description Language and architecture implementation language languages. Figure 7.1 highlights the transformations within the proposed solution.

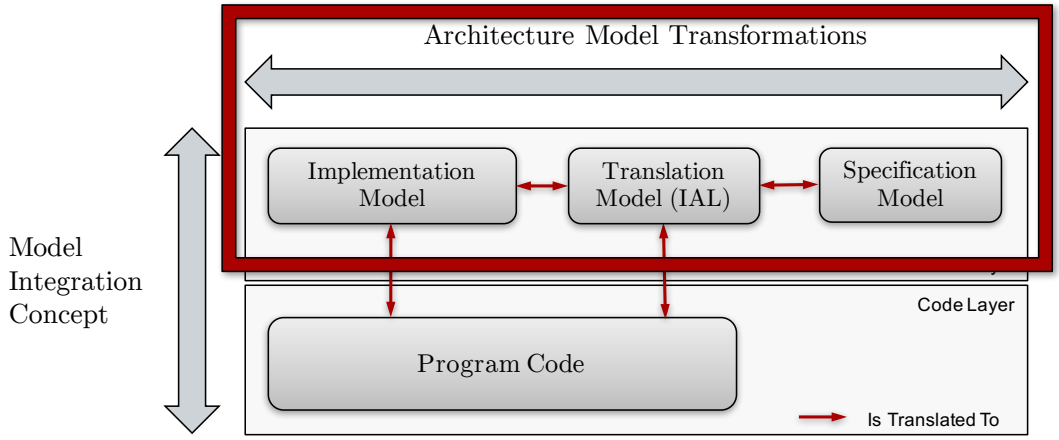


Figure 7.1: The architecture model transformations highlighted in the overview of the proposed solution

Two types of architecture model transformations are part of the Explicitly Integrated Architecture Process. First, transformations between architecture specification languages and the IAL as well as transformations between architecture implementation languages and the IAL are used to create a mapping between architecture specifications and implementations on a model level. Second, transformations within the IAL allow for translating between different related profiles of the IAL. The following sections present these types of transformations.

7.1 Transformations Between Specification or Implementation Languages and the Intermediate Architecture Description Language

The Explicitly Integrated Architecture Process translates between architecture specification and implementation languages. The IAL is used to reduce the number of transformations to define. Therefore translations between architecture specification and implementation languages and the IAL are defined. These transformations are exogenous, i.e. they translate models of one meta model into models of another meta model [MVG06].

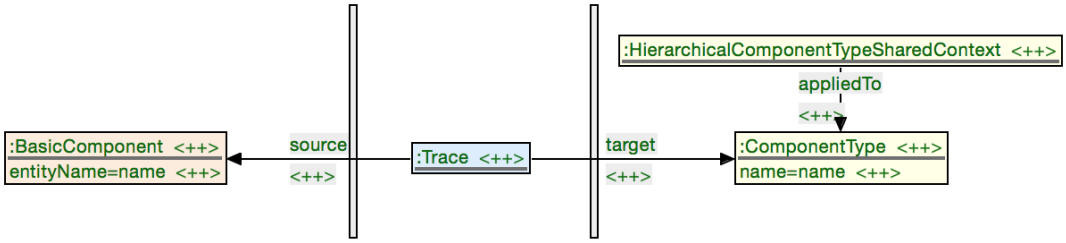


Figure 7.2: Example triple rule between the Palladio Component Model (left) and the IAL (right). The center is a correspondence graph, that relates source graph elements to target graph elements.

7.1.1 Example of a Transformation between an architecture language and the IAL

Transformations between architecture languages and the IAL can be defined in any model transformation technique that allows for exogenous transformations. In the implementation of the tool that accompanies this thesis, triple graph grammars (TGG) [Sch94] based on attributed, typed graphs [BET12], are used. In these typed graphs the graphs are models and the type graphs are meta models in the terms of this thesis. A TGG comprises triple rules, which declare how two graphs can be produced in alignment. Figure 7.2 gives an example of such a triple rule, which is used to translate bidirectionally between a *BasicComponent* in the Palladio Component Model (PCM) and a *ComponentType* in the IAL. It comprises a source graph, that is a model of the Palladio Component Model language, a target graph, that is a model of the IAL, and a correspondence graph, which is a model of a special correspondence language meta model, for relating source graph elements in the triple graph to target graph elements. The PCM model declares an object of the class *BasicComponent*, with an attribute *entityName*. The attribute is expected to have a value, that is caught in the variable *name*. The IAL model declares an object of the class *ComponentType* with an attribute *name*. The attribute is expected to have a value, that is named in the variable *name*. The IAL model declares that the stereotype *HierarchicalComponentTypeSharedContext* is applied to the component type. The object of the class *Trace* in the correspondence graph relates the *BasicComponent* to the *ComponentType*.

The triple rule is a basis from which operational rules are derived [HEGO10, SG14a]. Operational rules can be executed upon a given triple graph. A triple graph is a set of graphs: The source graph, the correspondence graph and the target graph. Forward translation, backward translation, correspondence check, and integration rules can be derived. Forward translation rules are used to create a target graph based on a source graph. Backward translation rules are used to create source graphs based on a target graph. Consistency check rules are used to check whether each element on each side has a correspondence on the other side via the correspondence graph. Integration rules create a correspondence graph based on given source and target graphs. Figure 7.3 shows an operational forward transformation rule which has been derived from the triple rule in Figure 7.2. The `<tr>` signs mean that a match is searched for this node or edge in the original triple graph. The `<++>` signs mean that these nodes or edges

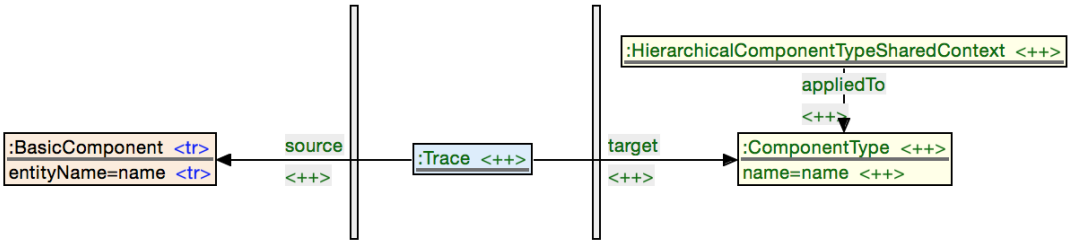


Figure 7.3: Example operational forward transformation rule between the Palladio Component Model (left) and the IAL (right), based on the triple rule in Figure 7.2.

are to be created.

Figure 7.2 declares that a graph comprising one node of the type *BasicComponent* in the source graph of the triple graph is to be translated. When such a subgraph is found in the source graph, nodes of the types *ComponentType* and *HierarchicalComponentTypeSharedContext* are created in the target graph with the given relations. The value of the attribute *name* of the *ComponentType* node is set to the value of the *entityName* to be translated.

7.1.2 Example of the Propagation of Changes between Models of an architecture language and the IAL

Forward and backwards transformations cannot be used for deleting nodes. To do so, a forward propagation or backwards propagation can be used as actions upon the triple graphs [SG14c, SG14b]. Such a propagation is a set of steps. During a forward propagation, first a consistency check is executed. All nodes and edges on the target graph, that have no corresponding element on the source graph as defined by the triple rules, are deleted. Then forward rules are executed for each node on the source graph that has no correspondence. As a result, changes in the source graph, including creations, changes, and deletions, are propagated to the target graph. The backwards propagation works likewise.

Figure 7.4 shows the execution of a forward propagation an example, based on the triple rule in Figure 7.2. Initially, the triple graph contains two PCM *BasicComponents* A, and B, which have corresponding *ComponentTypes* in the IAL. Then the basic component B is removed, and a new basic component C is created. The forward propagation removes the component type B from the IAL model and the dangling trace element. Then it creates a new IAL component type C with a corresponding trace element. The elements regarding component type A remain unchanged. All transformations defined in this thesis are available on the data medium attached to this thesis (see Appendix B).

7.2 Transformations Between IAL Profiles

The IAL comprises several profiles that are mutually exclusive (see Section 6.5). As an example, when an architecture is modelled with the IAL profile *Scoped Component Hierarchy* (see Definition 87) and, at the same time with the profile *Shared Context Component Hierarchy* (see Definition 88), this information would be inconsistent. An architecture cannot at the same

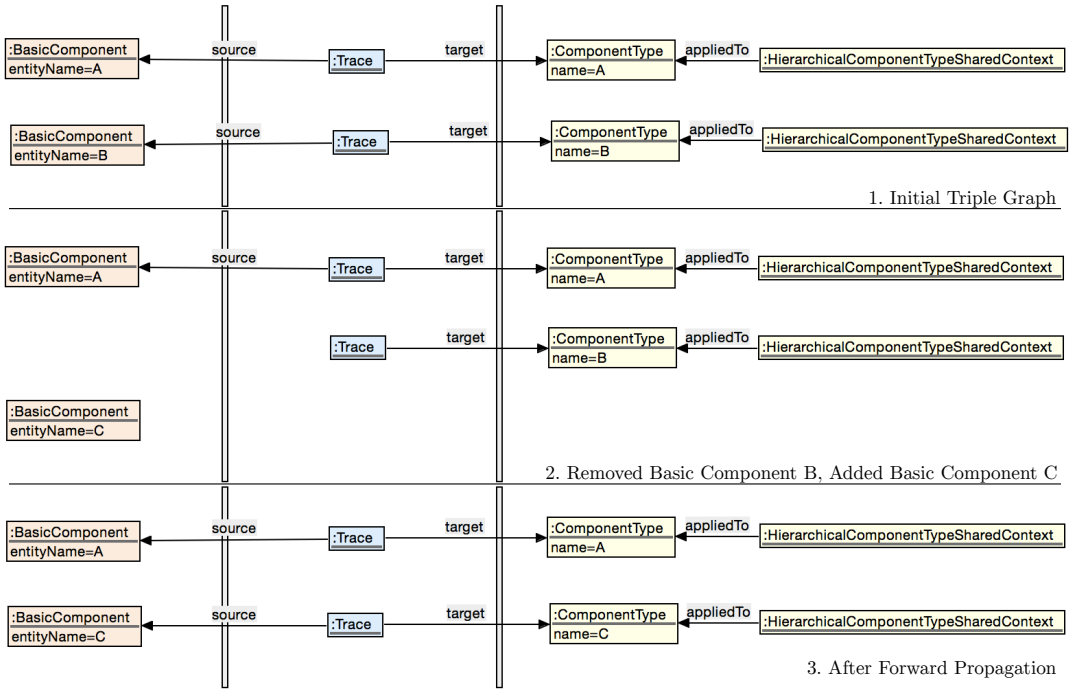


Figure 7.4: Example forward propagation between a Palladio Component Model model (left) and a model of the IAL (right), based on the triple rule in Figure 7.2.

time be a flat component type hierarchy and a deep component type hierarchy. Nevertheless, an architecture can be expressed in an architecture implementation language that defines scoped context component hierarchies, and should be viewed in an architecture language that can only model flat component hierarchies. To respect these situations, the Explicitly Integrated Architecture Process defines transformations between IAL profiles, which are called *inter-profile transformations* further on.

An inter-profile transformation is only possible for mutually exclusive profiles. E.g. a shared component architecture can be translated into a scoped component architecture, but the component hierarchy is not concerned with the interface type. Therefore a translation between a scoped component hierarchy and typed interfaces is not defined. This limits the number of inter-profile transformations that are necessary. Figure 7.5 shows the mutually exclusive profiles and transformations between them.

The inter-profile transformations are monotonic productions [Sch94]. This means that only new elements are created during the transformation. No information is lost, but it leaves inconsistent information in the model. This inconsistency is resolved by the language transformations between architecture languages and the IAL for specific languages. They only consider the information they require for the translation.

Figure 7.6 shows an example how these transformations are used within the Explicitly Integrated Architecture Process. In this example, an excerpt of an implemented architecture is modelled using the profile for scoped component type hierarchies (**step 1**). It contains two component types. The component type A has the component type B as child component using

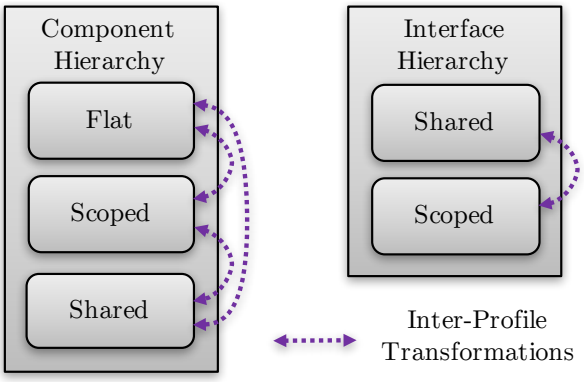


Figure 7.5: Transformations between IAL profiles. Each arrow shows between which profiles transformations have been specified.

the said profile. The model is translated into a view with a shared component type hierarchy (**step 2**), because the target language cannot express scoped hierarchies. This translation adds the shared context hierarchy, without removing the scoped hierarchy information. The targeted language is then used to change the component type *B*. The component type's name is changed to *C* (**step 3**). The profile Scoped Component Hierarchy is ignored in this step. When the architecture is translated into the source language again, the shared context component hierarchy is ignored. **Step 4** shows the result of the translation. The child component relation still exists, although the language that was used to change the model is not designed to handle child type relationships. In the following sections, the inter-profile transformations are described.

7.2.1 Component Hierarchy

All three profiles dealing with in the component hierarchy are mutually exclusive.

Scoped to Shared Component Hierarchy

Figure 7.7 shows the inter-profile transformation for translating architectures with the profile Scoped Component Hierarchy (see Definition 87) to use the profile Shared Context Component Hierarchy (see Definition 88). The example in Figure 7.6 is based on this transformation. The description is based on the model transformation tool *Henshin* [ABJ+10b], but other tools for in-place graph transformations are also usable. The transformation shown in Figure 7.7 converts an architecture with a scoped component hierarchy into an architecture with a shared context hierarchy. In the Scoped Component Hierarchy profile, component types own child component types and instances of child component types. The Shared Context Component Hierarchy profile declares all component types on the same level.

The execution is organized by the sequential unit called *main*. A sequential unit declares in which order rules are executed. When for any rule within a sequential unit no match is found, the translation ends. First the rule *architecture* is executed, which adds a stereotype *HierarchicalArchitectureSharedContext* to the architecture. The transformations are designed to be idempotent. I.e. they can be executed multiple times. They only add information when it

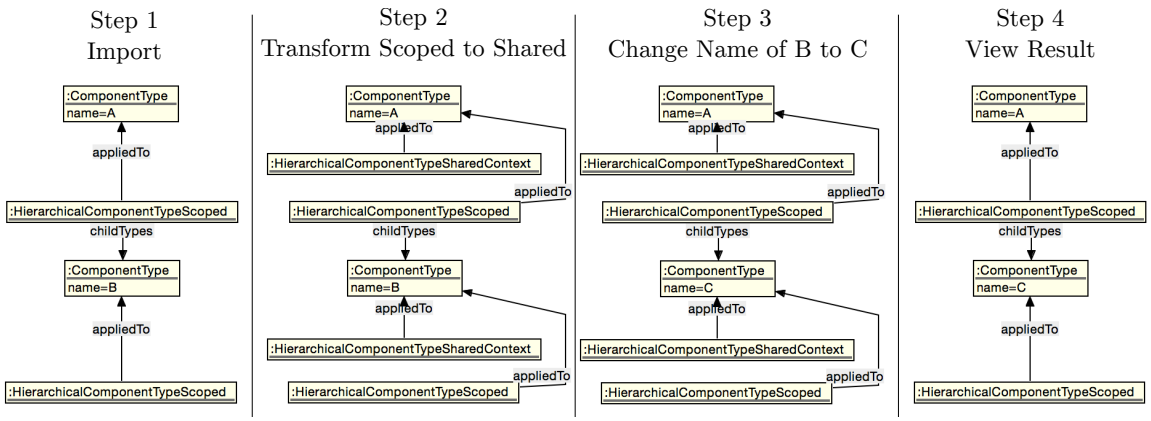


Figure 7.6: An excerpt of an architecture, that is transformed between the IAL profile Scoped Component Hierarchy and the profile Shared Context Component Hierarchy. A change within the shared context view is propagated without loosing information about the scoped component type hierarchy.

is missing. They will therefore not duplicate existing information. The rule *componentType* is a *multi-rule* [ABJ$^+$10a, BEE$^+$10], as denoted by the stars on the operators (*preserve**, *forbid**, *create**). Multi-rules are executed as often as there is a match to be found. The rule therefore adds the stereotype *HierarchicalComponentTypeSharedContext* to each component type. The rule *childInstances* contains nested multi-rules. It ensures that in both profiles, the child instances are set consistently. In the operators, a slash-separated path is appended to the star notation. Each path fragment symbolizes a nesting. I.e. the component type and the stereotypes are on the same level. This part of the rule is executed as often as a match can be found for these elements. When such a match is found, the rule containing the component instance is executed. For every component type which has both stereotypes attached, a child instance relationship is searched for in the *scoped* profile, and created consistently in the *shared* profile. The rule *sytemInstances* ensures that all system instances of the scoped hierarchy architecture are also set to be system instances in the shared context hierarchy architecture.

Shared to Scoped Component Hierarchy

Figure 7.8 shows the transformation from the profile of shared (see Definition 88) to the profile of scoped component hierarchies (see Definition 87). The sequential unit *main* organizes the execution. First, the rule *architecture* is used to ensure that a stereotype *HierarchicalArchitectureScoped* is applied to the architecture element. Then the rule *componentType* ensures that each component type has the stereotype *HierarchicalComponentTypeScoped*, so that it can define child component types within its scope. The rule *noParentMeansSystemType* makes sure that all types that have no parent type are marked as system types. Next the rule *childInstances* synchronizes the child instances between the two profiles. At last the rule *noParentMeansSystemInstance* declares all component instances that have no parent type to be system instances.

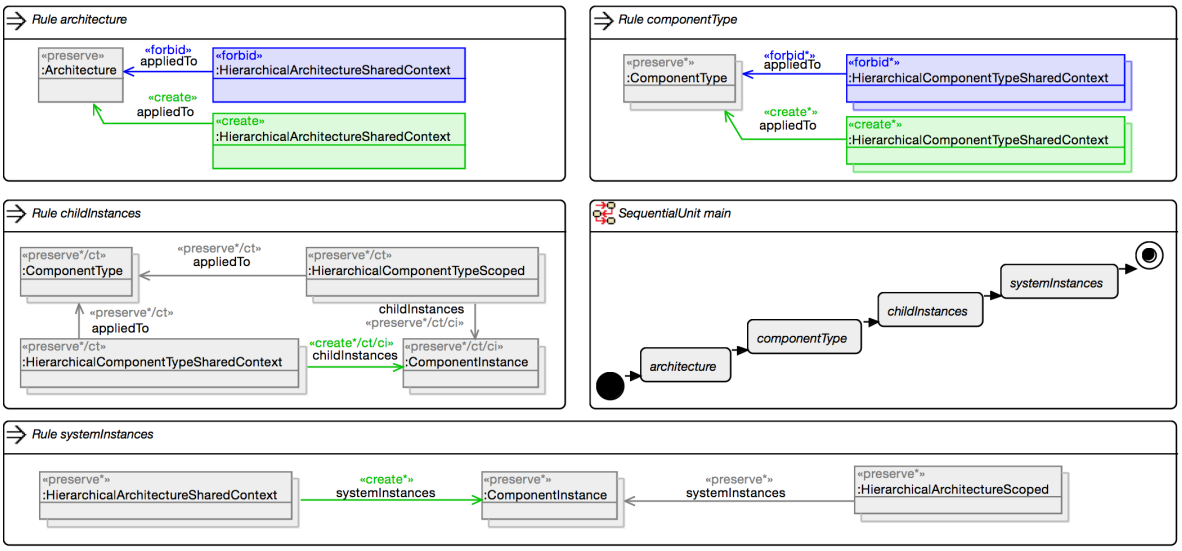


Figure 7.7: The inter-profile transformation scoped to shared component hierarchy

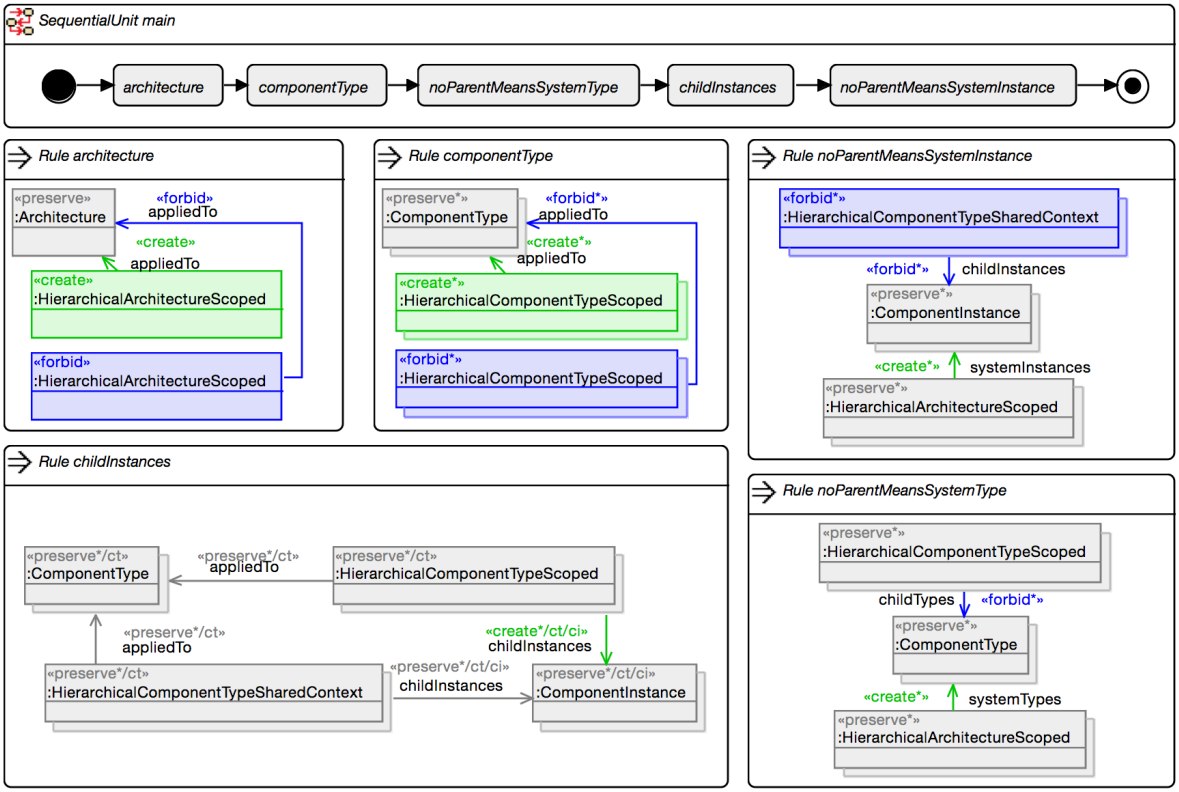


Figure 7.8: The inter-profile transformation shared to scoped component hierarchy

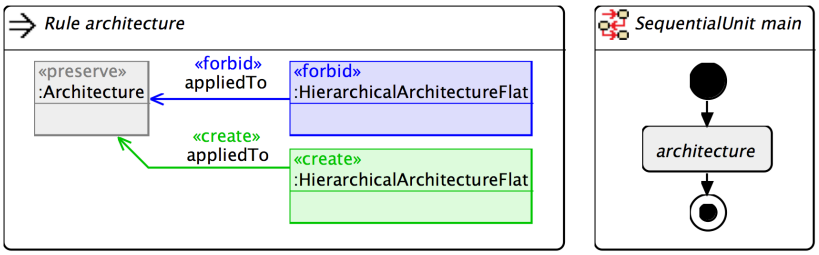


Figure 7.9: The inter-profile transformation shared to flat component hierarchy

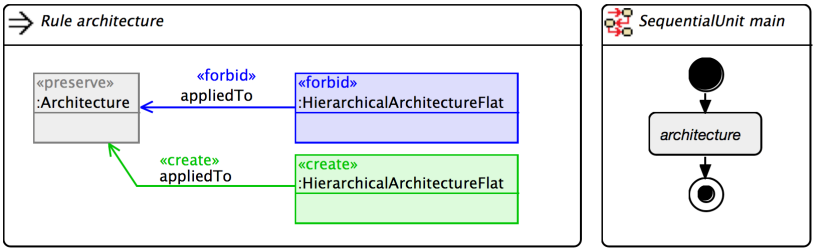


Figure 7.10: The inter-profile transformation scoped to flat component hierarchy

Shared to Flat Component Hierarchy

Figure 7.9 shows the transformation from the profile of shared (see Definition 88) to the profile of flat component hierarchies (see Definition 86). The sequential unit *main* organizes the execution. The only rule *architecture* applies the stereotype *HierarchicalArchitectureFlat* to the architecture.

Scoped to Flat Component Hierarchy

Figure 7.10 shows the transformation from the profile of scoped (see Definition 87) to the profile of flat component hierarchies (see Definition 86). The sequential unit *main* organizes the execution. The only rule *architecture* applies the stereotype *HierarchicalArchitectureFlat* to the architecture.

Flat to Shared Component Hierarchy

Figure 7.11 shows the transformation from the profile of flat (see Definition 86) to the profile of shared (see Definition 88) component hierarchies. The sequential unit *main* organizes the execution. First, the rule *architecture* applies the stereotype *HierarchicalArchitectureSharedContext* to the architecture element. The rule *componentTypes* adds the stereotype *HierarchicalComponentTypeSharedContext* to each component type. At last, the rule *systemInstances* marks all component instances as system instances that have no parent component type.

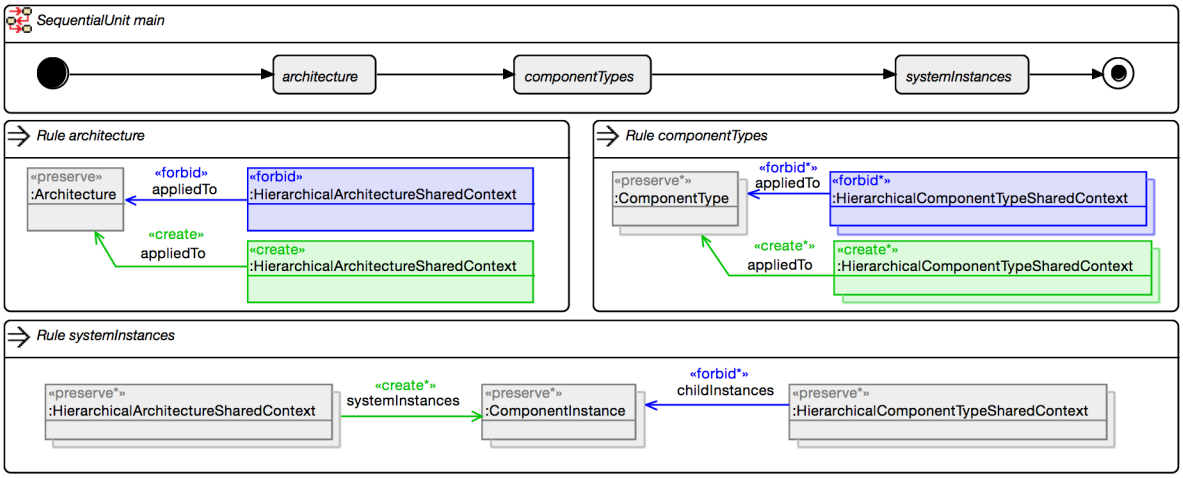


Figure 7.11: The inter-profile transformation flat to shared component hierarchy

Flat to Scoped Component Hierarchy

Figure 7.12 shows the transformation from the profile of flat (see Definition 86) to the profile of scoped component hierarchies (see Definition 87). The sequential unit *main* organizes the execution. First, the rule *architecture* applies the stereotype *HierarchicalArchitectureScoped* to the architecture element. The rule *componentTypes* applies the stereotype *HierarchicalComponentTypeScoped* to each component type. Next the rule *noParentMeansSystemType* marks every component type that has no parent type a system type. Analogously the rule *noParentMeansSystemInstance* marks every component instance that has not parent type a system instance.

7.2.2 Interface Hierarchy

The interface hierarchy profiles are mutually exclusive. An architecture cannot at the same time define interfaces as children of component types and interfaces in a shared repository on the same level.

Scoped to Shared Interface Hierarchy

Figure 7.13 shows the transformation from the profile of scoped (see Definition 85) to the profile of shared interface hierarchies (see Definition 84). The execution is also organized by the sequential unit *main*, which only executes the rule *architecture*. The rule creates a stereotype *SharedInterfacesArchitecture* and applies it to the architecture element, if none exists prior to the rule execution.

Shared to Scoped Interface Hierarchy

Figure 7.14 shows the transformation from the profile of shared (see Definition 84) to the profile of scoped interface hierarchies (see Definition 85). This transformation is more complex

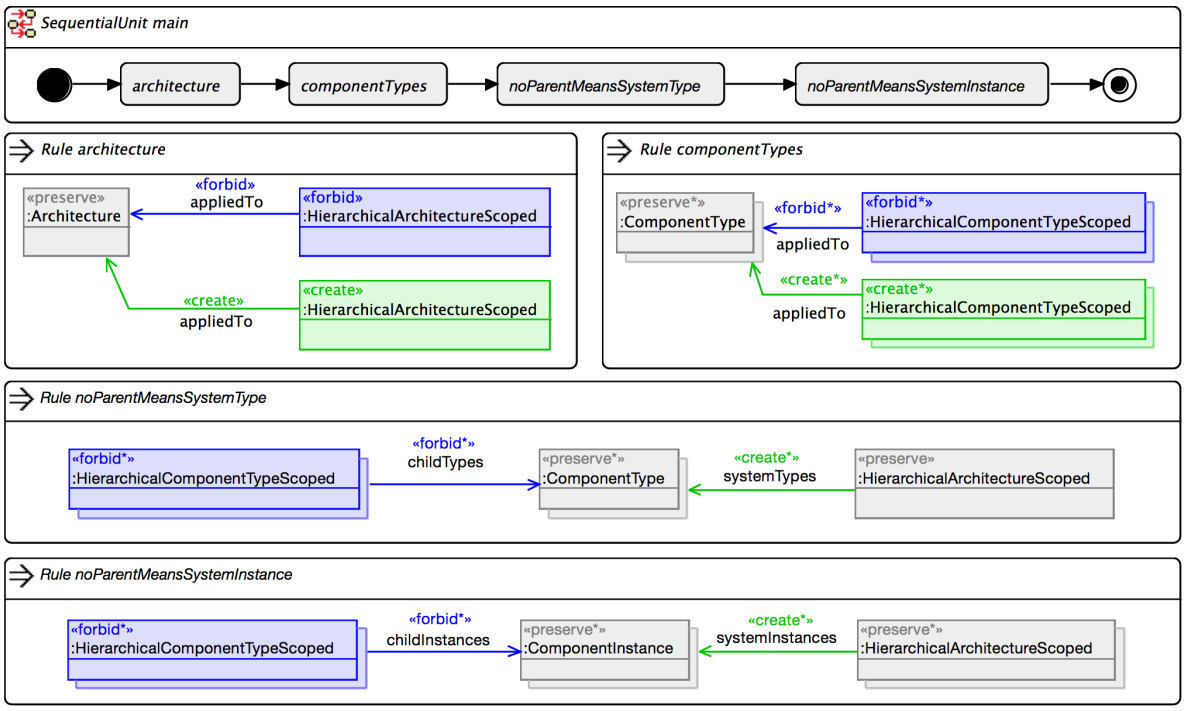


Figure 7.12: The inter-profile transformation flat to scoped component hierarchy

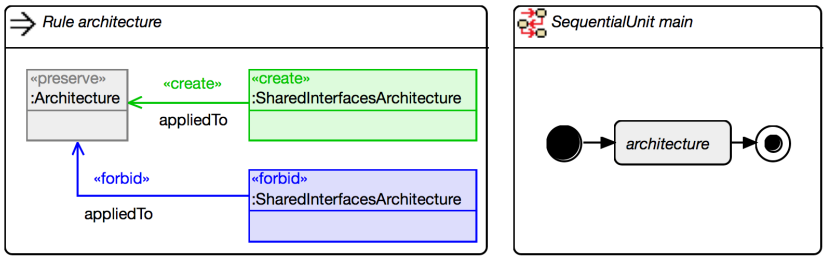


Figure 7.13: The inter-profile transformation from a scoped to a shared interface hierarchy

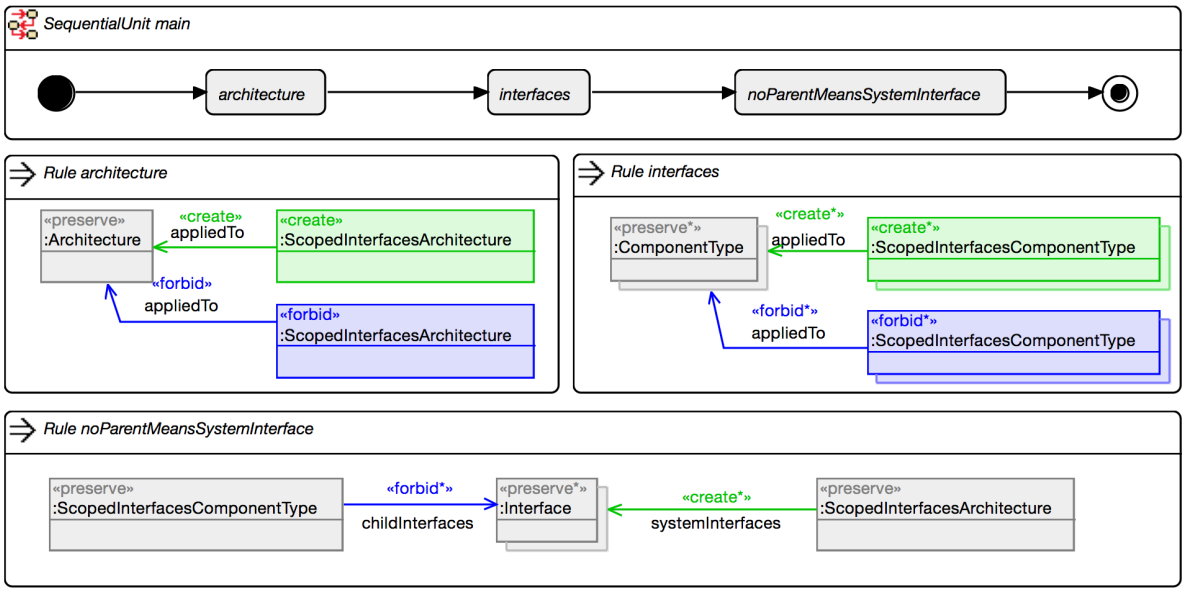


Figure 7.14: The inter-profile transformation from a shared to a scoped interface hierarchy

than the transformation in Section 7.2.2, because more elements and references have to be created. The sequential unit *main* organizes the execution. First, the rule *architecture* is used to ensure that a stereotype *ScopedInterfacesArchitecture* is applied to the architecture element. Then the rule *interfaces* ensures that each component type has the stereotype *ScopedInterfacesComponentType*, so that it can define interfaces within its scope. The rule *noParentMeansSystemInterface* marks all interfaces that are not child interfaces of a component type as system interfaces. After the transformation all interfaces are defined within the same scope. This is semantically equivalent to a shared interface hierarchy.

7.2.3 Other Profiles

Not all possible transformations make sense. It would be possible to translate between operation-based and event-based interfaces, connectors and data types. Operation-based and event-based handling of messages are alternative styles of messaging. There are two reasons for not providing this transformation: First, the analysis of an operation-based system that is modelled using events – or vice versa – would not be as reliable as it would be wished for. Second, this translation is not defined because the profiles are not mutually exclusive. Some architecture languages allow for both styles of communication. E.g. EJB uses Session Beans for operation-based communication and Message-Driven Beans for event-based communication. Consider an AIL, which allows for both operation-based and event-based communication, and a specification language that allows only for event-based communication. In an exemplary system, the translation would be processed as follows:

1. The AIL describes an operation-based interface. A translation of the code to the specification language is started.

2. In an inter-profile transformation, the operation-based interface is transformed to an event-based interface. The architecture specification now contains both stereotypes, event-based and operation-based, in the IAL.

3. The specification language only handles event-based communication. The interface is edited in this view. Afterwards the transformation is started from the specification language view to code.

4. No inter-profile transformation is triggered, because the target language handles both communication styles.

It is now unclear how the code should be structured: event-based or operation-based. Both stereotypes exist and can be handled by the architecture implementation language.

7.2.4 Profile Activation

When a translation between two architecture languages is executed, the transformations might expect information of specific profiles. E.g. a PCM transformation expects deployment information using the stereotype *ArchitectureWithDeploymentFragments*. When the source language does not provide deployment information, this stereotype is not applied to the architecture object.

The possible lack of information can either be handled by each transformation definition between the IAL and an architecture language, or by the process in the context of inter-profile transformations. To create a unified handling of missing stereotypes, in the approach at hand the latter variant has been chosen. Therefore, in the context of inter-profile transformations, additional transformations for adding missing stereotypes are executed. These transformations are called *profile activation transformations*.

During the translation between two architecture languages, it is known which profiles each language uses in the transformations towards or from the IAL. This information is used to determine which profile activation transformations need to be executed. In the example given above, the profile activation for the Deployment profile is executed. The transformation is shown in Figure 7.15. The transformation adds the stereotype *ArchitectureWithDeployment-Fragments* if it does not exist. These transformations exist for all profiles in optional categories (see Section 6.5), that apply stereotypes to the architecture. The transformations between the IAL and architecture languages can therefore rely on these stereotypes to be applied to the architecture object.

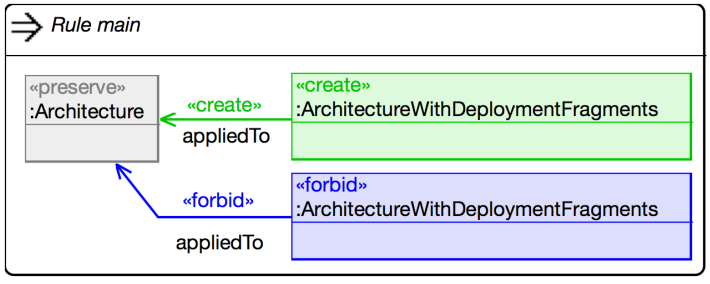


Figure 7.15: The profile activation transformation for the Deployment profile

7.3 Summary

This chapter described where architecture models have to be translated during the Explicitly Integrated Architecture Process. Bidirectional exogenous transformations translate between Architecture Description Languages, architecture implementation languages, and the IAL. Triple graph grammars are a good utility to implement these transformations. Inter-profile transformations translate between profiles of the IAL. This chapter described the different kinds of transformations and gave examples how transformations are defined and executed in the context of the proposed solution. The next chapter describes the use of the architecture model transformations and the other parts of the proposed solution in the Explicitly Integrated Architecture Process.

8 Explicitly Integrated Architecture Process

The Explicitly Integrated Architecture Process [Kon16] provides means to change architecturally relevant program code at design time with architecture specification languages, and automatically propagate the changes to the program code. Architecture model information must on the one hand be extracted from the code, and on the other hand integrated with the program code after changing or creating the specification model.

8.1 Process Overview

The Explicitly Integrated Architecture Process is visualized in Figure 8.1. It can be started either from program code that complies to an implementation model – including code that has not been developed using the process yet – or from either a new, or an extracted specification model. When the process is started from a specification model, it is assumed that the software is built from scratch by developing a specification model, or that the specification model has already been extracted from the program code using the process. When the process is started from the program code, it is assumed that the software is developed in compliance with an architecture implementation language, or has already been developed using the process, and therefore compliant program code exists.

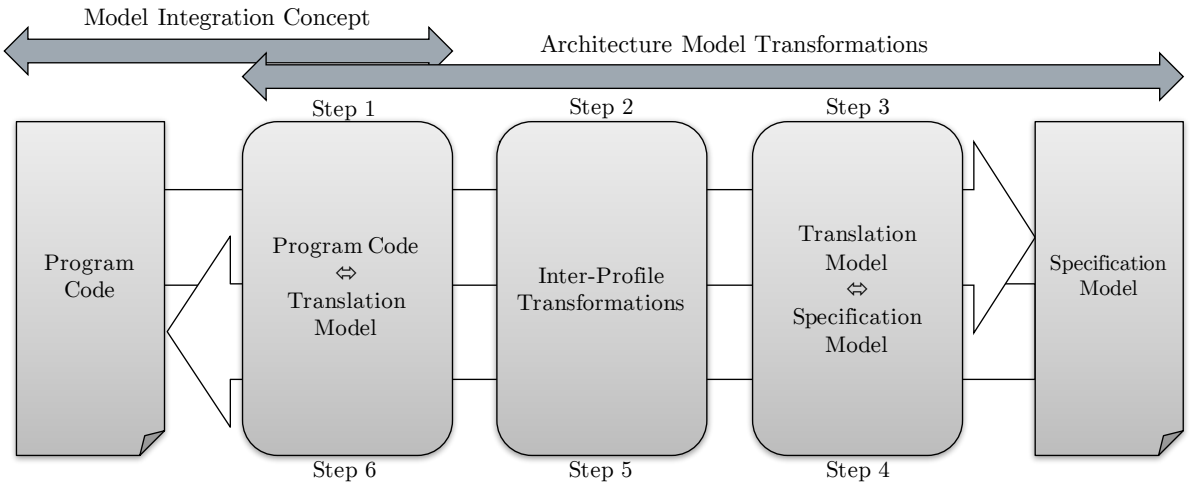


Figure 8.1: Overview of the Explicitly Integrated Architecture Process

8.2 Process Steps

The process defines three main steps for each direction. For extracting a specification model, the following steps are executed:

Step 1 extraction of a translation model from the program code via an implementation model;

Step 2 translation of the translation model according to the necessities of the involved languages, by executing inter-profile transformations;

Step 3 translation of the translation model into a specification model.

For integrating the specification model with the code, reverse steps are executed:

Step 4 translation of the specification model into a translation model;

Step 5 translation of the translation model according to the necessities of the involved languages, by executing inter-profile transformations;

Step 6 integration of the translation models with program code via an implementation model.

The following describes the steps with mode details. An example of the process in given in Section 4.4 on page 37. For this description, we assume that the process is started from existing program code.

1. *Program Code to Translation Model:* In the activity *Program Code to Translation Model* a translation model is created based on the program code. The activity comprises three subactivities, as shown in Figure 8.2. First, in the subactivity *Code to Implementation Model* the code is translated into an implementation model using the Model Integration Concept. Then, in the subactivity *Implementation Model to Translation Model*, a translation model is created from the implementation model using architecture model transformations. In the subactivity *Code to Translation Model* architecture information in the program code that cannot be expressed with the implementation model language is added to the translation model using the Model Integration Concept. The activity *Code to Translation Model* results in a translation model that represents the architecture of the code. These code-to-model translations are described in detail in Chapter 5. The Intermediate Architecture Description Language as language for the translation model is described in Chapter 6. The model-to-model translations are described in Section 7.1.

2. *Inter-Profile Transformations:* As shown in [Mü10], languages for software architecture have different kinds of information they are able to describe. The IAL handles these differences by splitting its meta model into different profiles. Depending on the target language to translate an IAL model into, some profiles are interpreted, while others are not. When the architecture implementation language and the architecture specification language in the process have such differences, it is necessary to translate between them. E.g. a flat component architecture can be represented in a hierarchical way with one hierarchical level. This is done in the activity *Inter-Profile Transformations*. The transformations do not delete the original information, therefore no information is lost. The IAL is described in Chapter 6. The inter-profile transformations are described in Section 7.2.

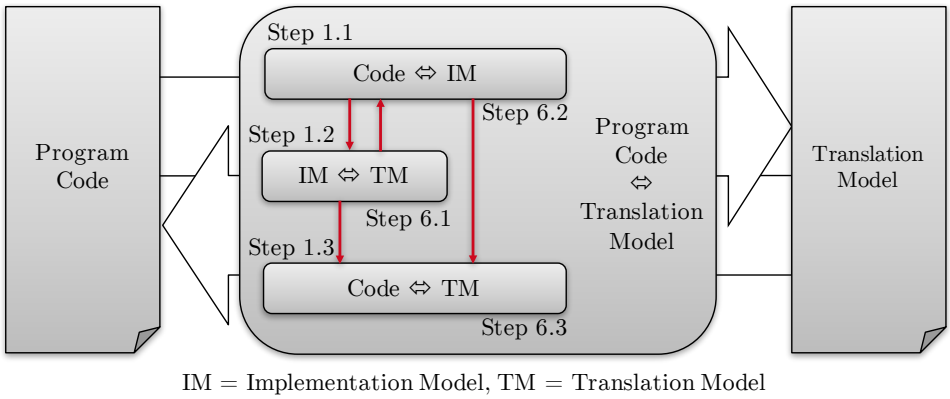


Figure 8.2: Subactivities of the steps 1 and 6 of the Explicitly Integrated Architecture Process

3. *Translation Model to Specification Model:* In this activity, the translation model is translated into a specification model. This translation is described in Section 7.1.

When the specification model is available, it can be viewed, analyzed, and changed with its original tools. The changed model can then be integrated with the existing program code as follows:

4. *Specification Model to Translation Model:* In this activity, the changes in the specification model are propagated to the translation model. This translation is described in Section 7.1.

5. *Inter-Profile Transformations:* The counterparts of the inter-profile transformations above are now executed, e.g. to flatten a hierarchical architecture. In this example, the information about the flat hierarchy remains untouched. The inter-profile transformations are described in Section 7.2.

6. *Translation Model to Code:* This activity comprises two subactivities. In the subactivity *Translation Model to Implementation Model*, the translation model is translated into an implementation model. In the subactivity *Translation Model & Implementation Model to Code* both models are taken as input to propagate the model changes to the program code. The integration is described in Chapter 5. The result of this activity is the program code that is changed according to the model changes in the specification model.

8.3 Summary

This chapter described the Explicitly Integrated Architecture Process. The process is used to integrate architecture model information with program code, extract this information, and create a temporary specification model view. Changes in this model can be propagated to the program code with the process.

9 Implementation

This chapter describes the implementation of tools in the context of the Explicitly Integrated Architecture Process. In Section 9.1 the running example of Section 5.6.1 is revisited. Then two tools are described using the running example: The tool *Codeling* executes the Explicitly Integrated Architecture Process (see Chapter 8). It supports the development of transformations between the code and the model representation of model notations, inter-profile transformations, and bidirectional architecture model transformations with libraries. As described in Section 5.7, several artefacts can be generated based on the formal definitions of integration mechanisms: meta model notation libraries, which contain the program code structures that represent meta models; model notation transformations, which translate between code and model views, based on the definition of notations of the Model Integration Concept (see Chapter 5); and execution runtime stubs, that use the generated program code to instantiate model elements in a running program. A *code generation tool* has been implemented to generate these artefacts. Codeling and the code generation tool share artefacts in an input/output relation. Figure 9.1 shows the artefacts that the implemented tools use as input and output.

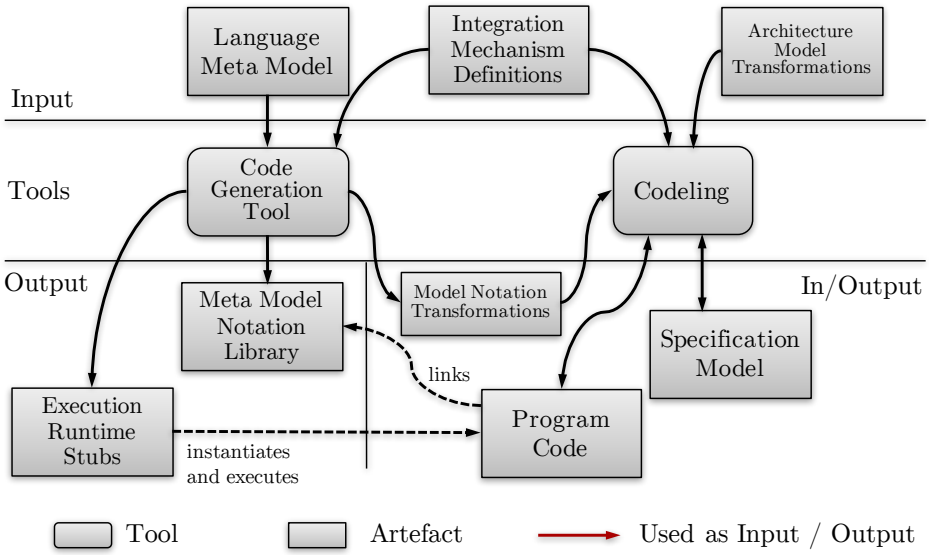


Figure 9.1: An overview of the tools and the artefacts they use as input and output

Codeling (see Section 9.2) is the tool for executing the Explicitly Integrated Architecture Process. It creates architecture specification model views upon program code, propagates changes in the model to the code representation, and can migrate program code from one architecture implementation language to another. Libraries in the context of Codeling support the development and execution of model notation transformations and architecture model

transformations, including inter-profile transformations.

The *code generation tool* (see Section 9.3) exploits the definition of integration mechanisms for the Model Integration Concept. The tool takes a language meta model as input. The tool's user maps integration mechanisms to elements of meta model elements. A library of abstract transformations and execution runtimes has been developed, which decreases the effort for creating transformations and execution runtimes for meta model elements, that are translated using the integration mechanisms. Based on the library of abstract transformations and execution runtimes for integration mechanisms, the code generation tool then generates a meta model notation library, model notation transformations, and execution runtime stubs. The integration of the generated code with Codeling is described in Section 9.3.7. A strategy for developing transformations is described in Section 9.4, before the chapter is summarized in Section 9.5.

9.1 Running Example

The following describes an example, that will be used as an illustrator for the functionality of the prototype tools. It is a simplified version of the running example in Section 5.6.1. In the running example, a simple software system is described with interconnected components and a state machine. Figure 9.2 show the meta model of the running example with elements for architectural structures on the upper side, and a state machine at the bottom. The structural part describes an architecture with components that have business operations. Components can reference other components. When a component has a reference to another component, it may invoke its operations. Component behaviour can be described with a state machine. A state machine contains a list of states and has an initial state. A state has a list of transitions that target a next state. All classes in the meta model but the architecture have an attribute *name*.

The example system (see Figure 9.3) is an excerpt of a store software, based on the CoCoME system [HKW+08]. It consists of two components *CashDesk* and *BarcodeScanner*. The bar code scanner has an operation *scanCode* to scan a bar code. The cash desk can be used to add items to a virtual shopping cart for billing. It therefor references the bar code scanner. It provides the operations *addItemToCart* and *checkout*.

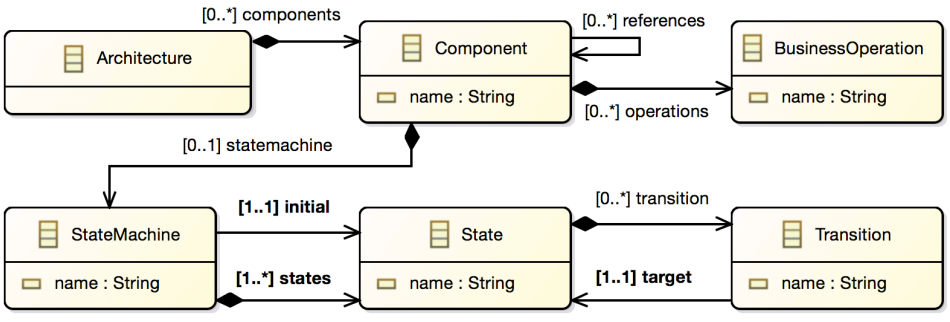


Figure 9.2: The structure and behavior meta model of the running example

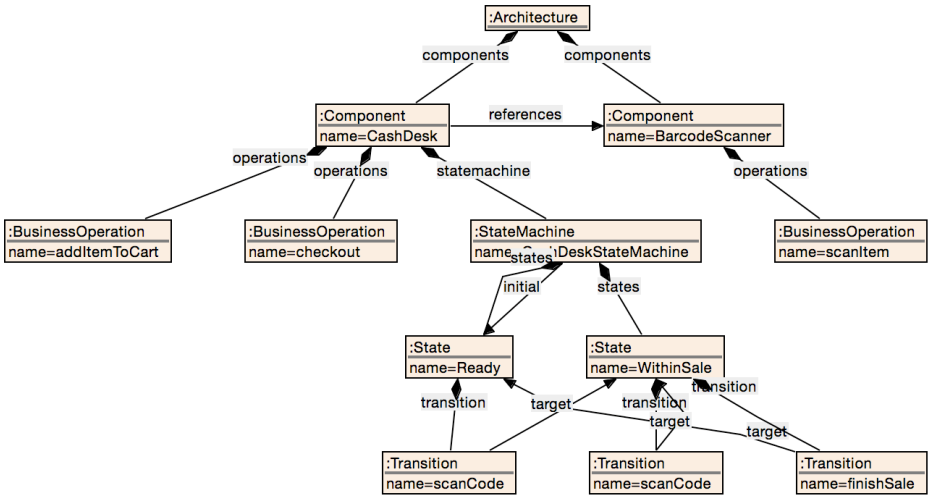


Figure 9.3: The implementation model representation of the original architecture in the running example

The behaviour of the cash desk is designed as a state machine. The initial state is *Ready*, which describes that the cash desk is ready for a new sale. When an item is scanned, it is in the state *WithinSale*. It remains in this state during subsequent scans until the sale is finished. Then it returns to the *Ready* state.

The code comprises representations of the specification elements, including implementation details. Components are translated using type declarations with an annotation `Stateful`, as declared by the EJB specification. This is close to the Type Annotation mechanism. Referenced components are notated with a field with the type that represents the targeted component as field type, and an annotation `EJB`, again, as declared by the EJB specification. This representation is close to the Annotated Member Reference mechanism. A state machine of a component is represented as a field with the annotation `Statemachine`. The type of the field is a representation of the state machine model element, following the Annotated Member Reference mechanism. Business operations are methods of components with the annotation `Operations`, as described by the Containment Operation mechanism. A state machine is notated with the Type Annotation mechanism, with states and the initial state translated with the Annotated Member Reference mechanism. A state has its transitions as containment operations. The transition's targets are translated via the Containment Operation Reference Annotation Parameter mechanism. Table 9.1 gives an overview of the mechanisms and notation details as a reference.

Listing 9.1 shows an excerpt of the code that implements the cash desk component with the Type Annotation notation. The package declaration and imports are not shown. The type `CashDesk` represents the component. It has an Annotated Member Reference field to the type that represents its state machine, and a representation of the reference to the bar code scanner. The `init` method is not part of the architectural description in the sense of this thesis. It initializes a runtime for the cash desk state machine. The annotation `PostConstruct` is described in the EJB specification. The method with this annotation is executed after the

Meta Model Element	Integration Mechanism or Notation Details
Architecture	**Ninja Singleton**
→ all attributes and references	Included in the owner's mechanism
Component	`Stateful` **annotation applied to a type (Type Annotation with specific annotation)**
→ references	`EJB` annotation applied to a member reference (Annotated Member Reference to Type Annotation or Static Interface with specific annotation)
→ statemachine	`Statemachine` annotation applied to a member reference (Annotated Member Reference to Type Annotation or Static Interface with specific annotation)
→ operations	Containment Operation for Types
StateMachine	**Type Annotation**
→ states	Annotated Member Reference to Marker Interface for x..* References
→ initial	Annotated Member Reference to Marker Interface for x..1 References
State	**Type Annotation**
→ transitions	Containment Operation for Types
Transition	**Contained in State.transitions**
→ target	Containment Operation Reference Annotation Parameter to Marker Interface for x..1 References
BusinessOperation	**Contained in Component.operations**

Table 9.1: The mapping of meta model elements and integration mechanisms for the implementation's running example

component is initialized. The business operation `addItemToCart` uses the reference to the bar code scanner to scan an item and adds the result to a list. It then triggers the transition *scanItem* in the state machine. The business operation `checkout` clears the list of items to simulate a successful sale, and triggers the *finishSale* transition in the state machine.

Listing 9.2 shows the cash desk's state machine in the Type Annotation notation. It owns two references in the Annotated Member Reference notation: The list of owned states and the initial state. In Listing 9.3 the state *WithinSale* is notated using the Marker Interface notation. It owns two transitions with the Containment Operation notation with their targets in the Containment Operation Reference Annotation Parameter notation.

9.2 Codeling - The Explicitly Integrated Architecture Process Tool

Codeling implements the Explicitly Integrated Architecture Process within the Eclipse IDE. It can therefore be used to create architecture specification language views upon program code, propagate changes in the specification model to the underlying program code; and to migrate program code that complies to an architecture implementation language to another architecture implementation language. Therefor Codeling requires specific transformation definitions for architecture implementation and specification languages. The following sections first describe a use case for Codeling in the running example (Section 9.2.1). Then the architecture is described in Section 9.2.2. Section 9.2.3 describes the implementation of the Explicitly Integrated Architecture Process in Codeling. Section 9.4 describes a strategy for developing transformations for codeling. At last, the extensibility of Codeling is described in Section 9.2.5.

```
@Stateful
public class CashDesk {
  final LinkedList<String> items = new LinkedList<>();
  StateMachineRuntime<CashDeskStateMachine> smr;

  @Statemachine
  CashDeskStateMachine cashDeskStateMachine;

  @EJB
  BarcodeScanner barcodeScanner;

  @PostConstruct
  public void init() throws IntegratedModelException {
    //... initializing the state machine runtime
  }

  @Operations
  public void addItemToCart() throws IntegratedModelException {
    items.add(barcodeScanner.scanItem());
    smr.executeTransition("scanCode");
  }

  @Operations
  public void checkout() throws IntegratedModelException {
    items.clear(); // Execute a real sale
    smr.executeTransition("finishSale");
  }
}
```

Listing 9.1: The original implementation of the component *CashDesk* in the running example

```
@StateMachine
public class CashDeskStateMachine {
  @Initial(Ready.class)
  State initial;

  @States({Ready.class, WithinSale.class})
  State[] states;
}
```

Listing 9.2: The original implementation of the state *CashDeskStateMachine* in the running example

```
public class WithinSale implements State {
  @Transition(target = WithinSale.class)
  public void scanCode() {
    Logging.log(LogLevel.MODEL, "Executing␣transition␣'scanCode'␣from␣state"+
      "'WithinSale'.");
  }

  @Transition(target = Ready.class)
  public void finishSale() {
    Logging.log(LogLevel.MODEL, "Executing␣transition␣'finishSale'␣from␣state␣'
        WithinSale'.");
  }
}
```

Listing 9.3: The original implementation of the *WithinSale* state in the running example

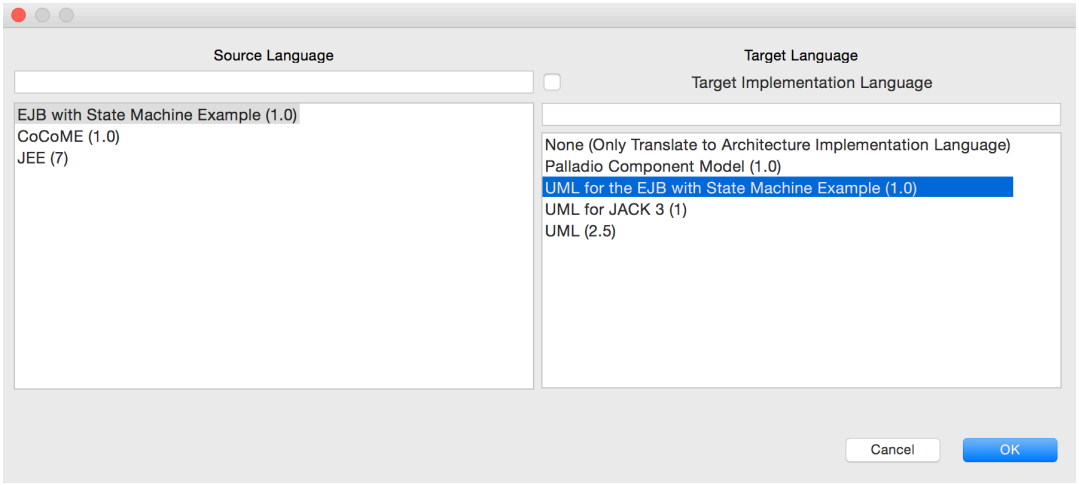


Figure 9.4: Selecting the architecture implementation and specification language for the translation in Codeling

9.2.1 Use Case - Evolving a Model of the Running Example

To start the process, Codeling provides a menu item "Start Explicitly Integrated Architecture Process" in the Eclipse IDE in the context menu of a project.

Upon using the menu item, the user has to choose the architecture implementation language, that the underlying code complies to, and the architecture specification or implementation language to translate it into. Figure 9.4 shows the selection dialogue. In the running example, the selected languages are *EJB with State Machine Example* and *UML for the EJB with State Machine Example*. These language definitions have been specifically implemented for this example. The code is then translated into the selected specification language. In-between, a set of intermediate models is created and transformed, following the Explicitly Integrated Architecture Process. The code, transformations, and extracted models for the running example are available on the data medium attached to this thesis (see Appendix B).

In **step 1.1** of the Explicitly Integrated Architecture Process, the model representation of the code is created using the generated transformations based on the integration mechanism (where applicable) and individually developed transformations, where generated transformations do not fit. The notation for components and their references do not exactly follow existing mechanisms. Their representation is taken from the EJB specification. The notation for components implies an annotation on a type declaration, just as it is declared in the Type Annotation mechanism. However, the annotation name `Stateful` does not match the annotation name `Component` required by the mechanism. The extracted model is shown in Figure 9.3 on page 217.

During this step, the notation's main program code element, the type declaration in the case of the Type Annotation mechanism, and the corresponding model element of the architecture implementation language are stored in an ID registry for later use. This is necessary for identifying when a model element is changed in an identifying part. E.g. when in the model the component name is changed, it must be known which type represented the component before the change, so that the type can be renamed. Listing 9.4 shows the corresponding entry

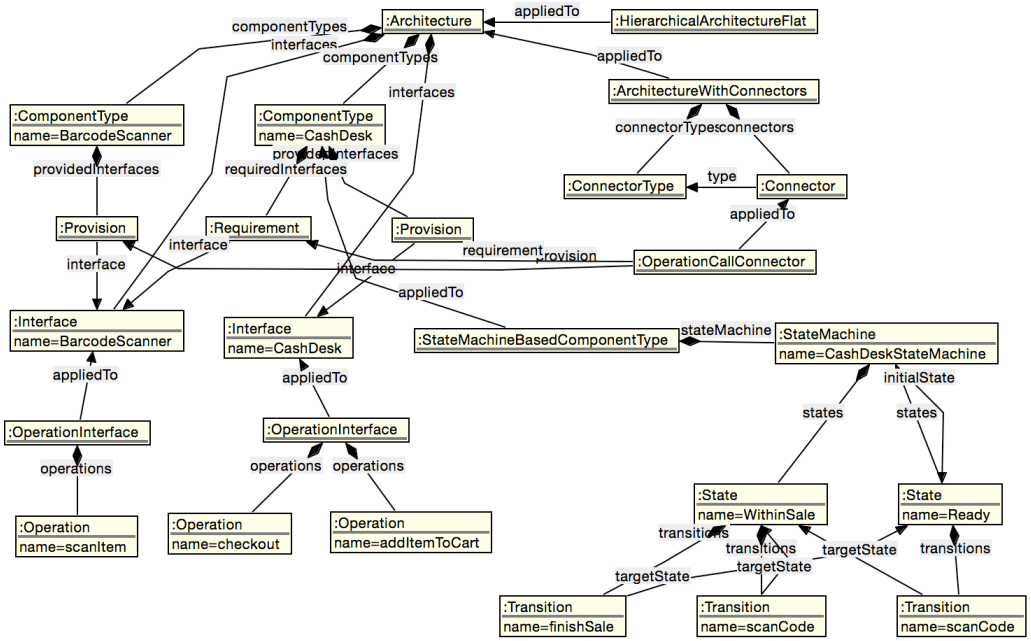


Figure 9.5: The original translation model of the architecture in the running example

in the registry file. It contains four columns, separated by a $ sign. The first column states an unambiguous identifier for the element, throughout the complete process. The second column is an unambiguous identifier for the code element. The third column contains an unambiguous identifier for the implementation model element, which uses the location of the element in the model as identifier. This is necessary, because not all meta models declare an identifying attribute for all classes. The fourth column is currently empty. It will contain an unambiguous identifier for the translation model element in the next step.

```
ibuadz878a7sdaduh$runningExample>org.codeling.example.CashDesk$/0/@components.0$          1
```

Listing 9.4: The ID registry entry for the translated *CashDesk* component after step 1.1

In **step 1.2**, the implementation model is translated into a translation model using the IAL (see Chapter 6). Figure 9.5 shows the representation of the architecture in the intermediate language. The ID registry entry is extended with an identifier for the newly created translation model element. Listing 9.5 shows the extended ID registry entry.

```
ibuadz878a7sdaduh$runningExample>org.codeling.example.CashDesk$/0/@components.0$/0/          1
    @componentTypes.0
```

Listing 9.5: The ID registry entry for the translated *CashDesk* component

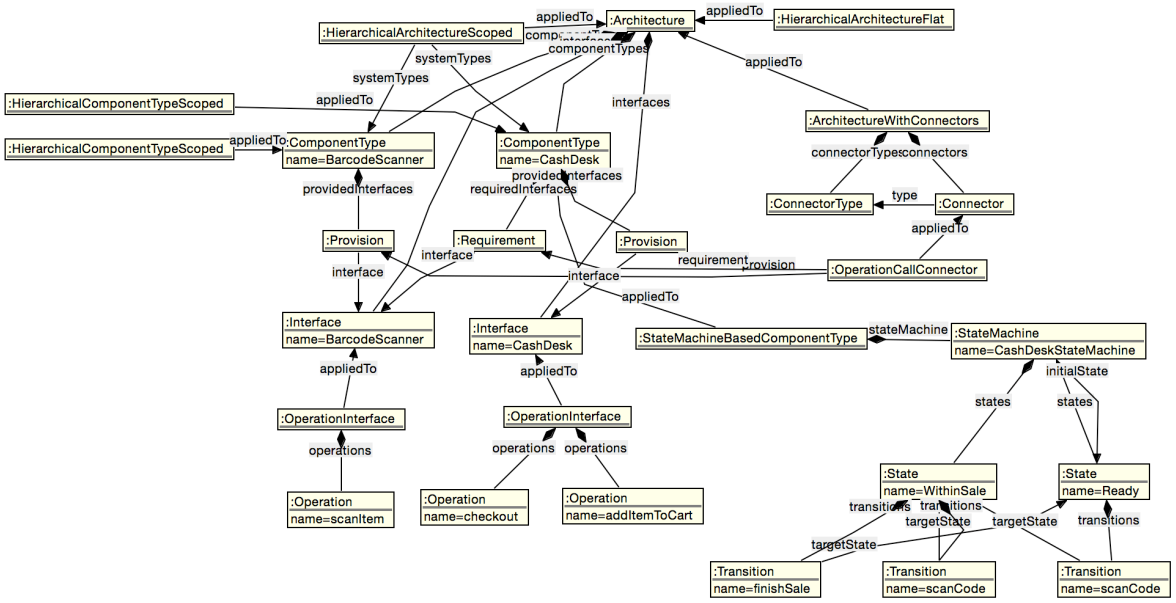


Figure 9.6: The original translation model of the architecture in the running example with component type hierarchy information

In **step 1.3**, component type hierarchy information is translated from the program code directly into the translation model. In the running example, no such information is included at the moment.

In **step 2**, the inter-profile transformations prepare the translation model for a UML representation. UML allows to describe hierarchical component type architectures. Figure 9.6 shows the representation of the architecture in the intermediate language after the inter-profile transformations. The new elements in the upper left side declare the scoped hierarchy definition.

In **step 3**, the translation model is translated into the specification model. In the example this is a UML model. UML diagrams based on this model is shown in Figure 9.7. A component diagram shows the components, their interfaces, and how they are interconnected. A UML state chart shows the state machine.

To follow the model elements during changes, the specification language must provide a mechanism to relate an ID to the element. In the running example, the ID is stored as structured comments attached to each element. The comments are not shown in the figure. In the running example the following changes are made to the architecture.

(1) The bar code scanner is declared to be a child component of the cash desk component.

(2) A new operation *closeDesk* is declared for the cash desk.

(3) The state chart is changed to include a new state *AwaitingPayment*.

(4) The transition *finishSale* now targets the new state instead of *Ready*.

(5) A new transition *paymentReceived* is introduced from *AwaitingPayment* to *Ready* instead.

Figure 9.8 shows the changed UML diagrams. On saving the modelling file, the specification

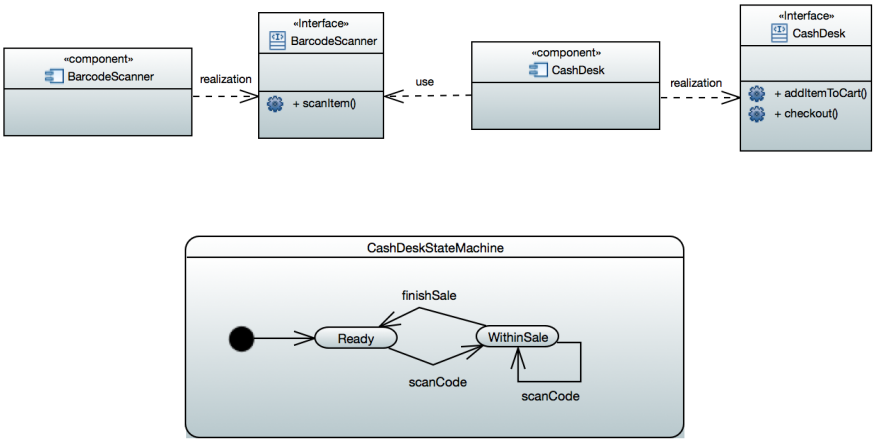


Figure 9.7: The original specification model of the architecture in the running example

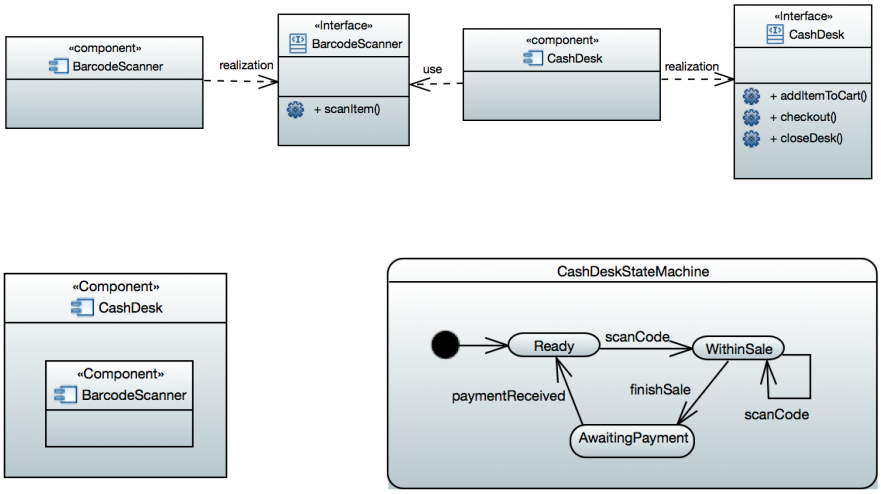


Figure 9.8: The changed specification model of the architecture in the running example

model representation is translated into the code representation automatically.

In **step 4**, the changes in the model are propagated to the former translation model of step 2 (see Figure 9.9). **(1)** The component type *CashDesk* now has the component type *BarcodeScanner* as child component type. The bar code scanner is now not a system type of the architecture element anymore. **(2)** The operation interface stereotype of the interface *CashDesk* now has a new operation *closeDesk*. The state machine now has **(3)** a new state and **(5)** transition. **(4)** The transition *finishSale* has the new state as target.

The model element identifiers in the ID registry entries use the model structure to identify elements. When an element is moved within this structure, the identifier is not valid anymore. Therefore the ID registry entry is updated during this step. In Listing 9.6 the identifier in the third column is changed to reflect such a movement of a translation model element. In Listing 9.5 the identifier was `/0/@componentTypes.0`, identifying the first model root `/0`, then the first child of that model root of the containment reference *componentTypes* `@componentTypes.0`. After the change, the identifier points to the second component type `@componentTypes.1`, because it has been moved internally during the translation process.

```
ibuadz878a7sdaduh$runningExample>org.codeling.example.CashDesk$/0/@components.0$/0/        1
    @componentTypes.1
```

Listing 9.6: The ID registry entry for the translated *CashDesk* component after step 4

In **step 5**, inter-profile transformations are executed to prepare the translation model for the architecture implementation language. This would add information about the flat architecture to the model. In the running example no changes are applied, because the information already exists.

In **step 6.1**, the translation model is translated into a representation in the architecture implementation language (see Figure 9.10). Compared to the original architecture in Figure 9.5, the changed implementation model includes the changes made in the specification language. The component type hierarchy is missing, because the implementation language is unable to express component type hierarchies.

Analogously to step 4, in this step model elements can be moved within the model structure. The ID registry entry is adapted accordingly. Listing 9.7 shows that the implementation model element moved from the first child `/0/@components.0` to the third `/0/@components.2`.

```
ibuadz878a7sdaduh$runningExample>org.codeling.example.CashDesk$/0/@components.2$/0/        1
    @componentTypes.1
```

Listing 9.7: The ID registry entry for the translated *CashDesk* component after step 6.1

In **step 6.2**, the architecture implementation model is translated into program code. Therefor the transformations of step 1.1 are executed in the opposite direction: towards the code. For each model element the transformations evaluate whether a code structure exists, that represents this model element. For this evaluation, an ID registry entry for the implementation model element is searched. If no such code structure exists, a corresponding code structure is

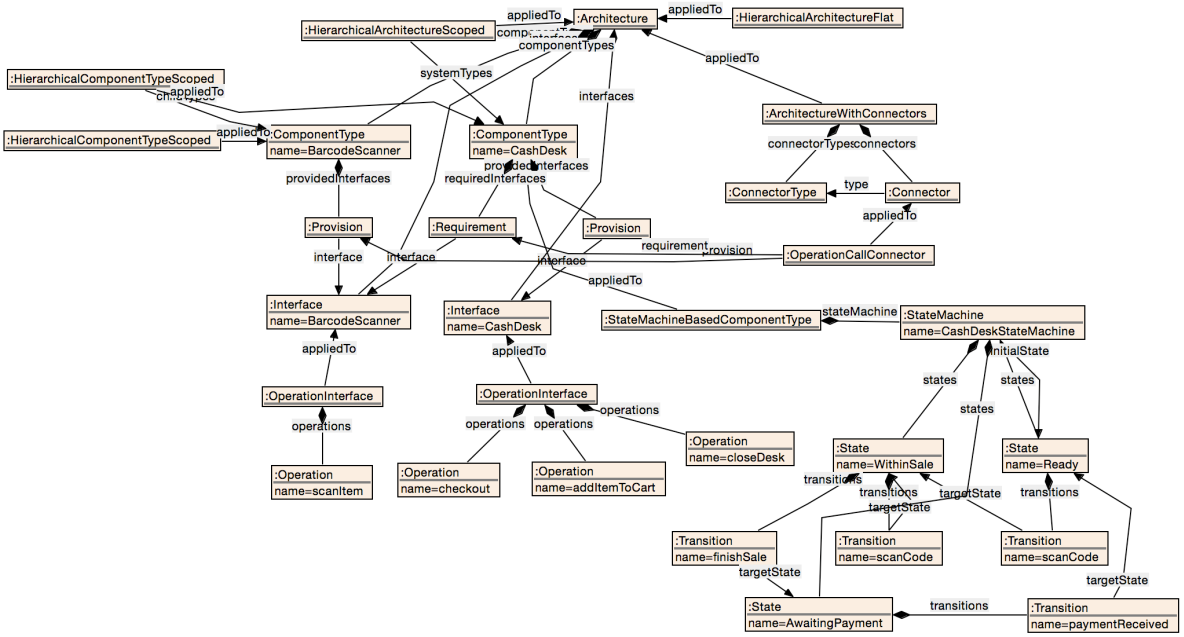


Figure 9.9: The translation model of the changed architecture in the running example

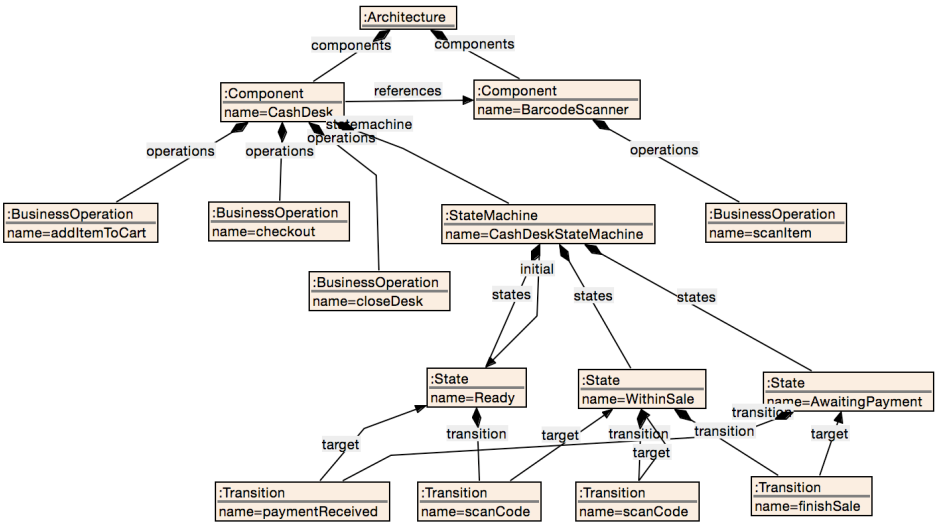


Figure 9.10: The architecture implementation model of the changed architecture in the running example

created. If a corresponding core structure already exists, it is updated if the model element has been changed. When a model element has been deleted, the corresponding code structure is deleted. Listing 9.8 shows the cash desk state machine type after the changes. Compared to the original type (Listing 9.2), the annotation of the *states* representation additionally declares a reference to the new state *AwaitingPayment*.

```
@StateMachine                                                                    1
public class CashDeskStateMachine {                                             2
  @Initial(Ready.class)                                                          3
  State initial;                                                                 4
                                                                                 5
  @States({Ready.class, WithinSale.class, AwaitingPayment.class})                6
  State[] states;                                                                7
}                                                                                8
```

Listing 9.8: The changed implementation of the *CashDeskStateMachine* in the running example

Listing 9.9 shows the changed cash desk type. In comparison to the original architecture, it includes the new business operation *closeDesk*.

```
@Stateful                                                                        1
public class CashDesk {                                                          2
  final LinkedList<String> items = new LinkedList<>();                           3
  StateMachineRuntime<CashDeskStateMachine> smr;                                 4
                                                                                 5
  @Statemachine                                                                  6
  CashDeskStateMachine cashDeskStateMachine;                                     7
                                                                                 8
  @Child                                                                         9
  @EJB                                                                           10
  BarcodeScanner barcodeScanner;                                                 11
                                                                                 12
  @PostConstruct                                                                 13
  public void init() throws IntegratedModelException {                           14
    //... initializing the state machine runtime                                 15
  }                                                                              16
                                                                                 17
  @Operations                                                                    18
  public void addItemToCart() throws IntegratedModelException {                  19
    items.add(barcodeScanner.scanItem());                                        20
    smr.executeTransition("scanCode");                                           21
  }                                                                              22
                                                                                 23
  @Operations                                                                    24
  public void checkout() throws IntegratedModelException {                       25
    // Execute a real sale                                                       26
    items.clear();                                                               27
    smr.executeTransition("finishSale");                                         28
  }                                                                              29
                                                                                 30
  @Operations                                                                    31
  public void closeDesk(){}                                                      32
}                                                                                33
```

Listing 9.9: The changed implementation of the component *CashDesk* in the running example

The hierarchy between the cash desk component type and the bar code scanner component

type are translated into code in **step 6.3**. In this step, information from the translation model, which cannot be expressed in the architecture implementation language, is translated into the code. Listing 9.9 shows the annotation `Child` on the field `barcodeScanner`, which expresses the component type hierarchy between the two interrelated component types.

The transition *finishSale* has a new target. This is expressed in the code as a changed annotation parameter as shown in Listing 9.10.

```
public class WithinSale implements State {
  @Transition(target = WithinSale.class)
  public void scanCode() {
    Logging.log(LogLevel.MODEL,"Executing␣transition␣'scanCode'␣from␣state␣'WithinSale'.");
  }

  @Transition(target = AwaitingPayment.class)
  public void finishSale() {
    Logging.log(LogLevel.MODEL,"Executing␣transition␣'finishSale'␣from␣state␣'WithinSale'.");
  }
}
```

Listing 9.10: The implementation of the changed *WithinSale* state in the running example

Listing 9.11 shows the program code of the Marker Interface model notation for the newly created state *AwaitingPayment* with the transition *paymentReceived* notated as Containment Operation, and the target *Ready* notated as Containment Operation Reference Annotation Parameter.

```
public class AwaitingPayment implements State {
  @Transition(target = Ready.class)
  public void paymentReceived() {}
}
```

Listing 9.11: The implementation of the new *AwaitingPayment* state in the running example

9.2.2 Architecture

This section gives an overview of the structural architecture of Codeling and its components and libraries to support the development of plugins for architecture implementation and specification languages. The structural architecture (see Figure 9.11) of Codeling comprises the following components:

User Interface The user interface of Codeling contains a menu item *Start Explicitly Integrated Architecture Process* in the Eclipse *Project Explorer* view to trigger the Explicitly Integrated Architecture Process. The menu item can be used, when a set of projects in the IDE are selected. The selected projects are the input for the translation process. Upon using the menu item, a language selection dialogue opens to choose the source and the target language of the translation (see Figure 9.4). On the left side of the dialogue, the architecture implementation language has to be chosen, in which the selected projects are implemented. The dialogue shows all registered implementation languages with their name and version in this box. On the right side of the dialogue, the target language can be chosen. The dialogue initially shows all registered specification language with their name and version. It is also possible to select the entry *None*, in which case only step 1 (Program Code to Implementation Model) of the process is executed. To execute an

implementation migration, the check box *Target Implementation Language* can be activated. The list of target languages then offers all registered architecture implementation languages.

Language Registry The language registry component collects all language definitions available in the system. It contains the interface `ILanguageRegistry` and abstract types as the basis for declaring architecture implementation or specification language definitions. The interface contains operations for adding language definitions to the registry, and for receiving registered languages.

Language Plugins Language plugins are components that provide specific language definitions for architecture implementation or specification languages, and register them at the language registry. These language implement specific architecture model transformations for architecture implementation languages, or model notation transformations for architecture implementation languages. A set of libraries has been developed in the context of Codeling, to support the development of language plugins. The library *Java Transformations* supports the development and execution of bidirectional model-code transformations for the Java programming language. The libraries *Henshin* and *HenshinTGG* support the execution of Henshin and HenshinTGG transformations.

Inter-Profile Transformations The Inter-Profile Transformation component provides the interface `IInterProfileTransformations` as means for executing inter-profile transformations. It can also determine, which inter-profile transformations (including profile activation transformations) should be executed, based on the source and the target languages of the translation.

Transformation Manager The Transformation Manager coordinates the execution of the Explicitly Integrated Architecture Process. It provides an interface `ITransformationManager` to trigger the translation of program code into the architecture languages and back. It uses language plugins to execute the architecture model transformations. For executing inter-profile transformations, it uses the corresponding component.

The following libraries are used throughout the whole tool. They are not shown in Figure 9.11:

IAL Meta Model The IAL meta model contains an implementation of the Intermediate Architecture Description Language (see Chapter 6) using Ecore for the language's kernel and EMF Profiles for the profiles.

Utilities The utilities contain functionality used by most other libraries and components, such as a common interface for logging within the application.

The following libraries are the basis for the implementation and execution of language plugins and transformations. In Figure 9.11 these libraries have a name starting with *Language Base*. They are describe in detail in Section 9.2.3.

Model Integration Concept The Model Integration Concept library contains an abstract type for executing bidirectional model-code transformations in the context of Codeling.

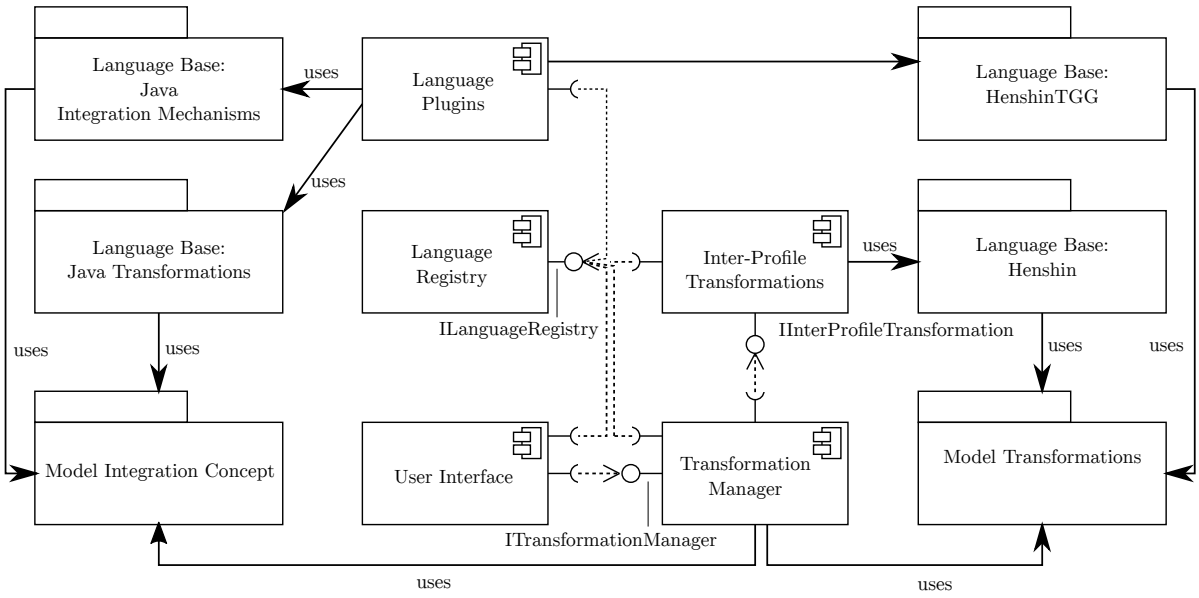


Figure 9.11: An overview of the components and libraries of Codeling

Java Integration Mechanisms The Java Integration Mechanisms library contains definitions of an excerpt of the integration mechanisms described in Section 5.6, implemented for the Java programming language.

Java Transformations The Java Transformations library contains abstract types and utilities for implementing model-code transformations based on the Java programming language. It also implements abstract transformations for notations based on integration mechanisms.

Model Transformations The Model Transformations library contains an abstract type for executing model transformations in the context of Codeling.

Henshin The Henshin library contains an API for executing Henshin transformations in the context of Codeling.

HenshinTGG The HenshinTGG library contains an API for executing HenshinTGG transformations in the context of Codeling, and an abstract type for defining language plugins with architecture model transformations based on HenshinTGG.

Table 9.2 gives an overview of mappings between the components and libraries states in this section, an the code artefacts that contains them in the Codeling program code.

9.2.3 Implementation Details

This section describes the structure and behaviour of Codeling in the context of the use case shown in Section 9.2.1, where a simple EJB implementation of a cash desk with a bar code scanner is translated to UML. The specification is changed in the model view, resulting in the

Component / Library	Code Artefact
Component: User Interface	Package `org.codeling.ui` in the project `org.codeling.core`
Component: Language Registry	Package `org.codeling.languageregistry` in the project `org.codeling.core`
Component: Language Plugins	Multiple projects in the folder `Language Integration`
Component: Inter-Profile Transformations	Package `org.codeling.interprofile` in the project `org.codeling.core`
Component: Transformation Manager	Package `org.codeling.transformationmanager` in the project `org.codeling.core`
Library: Utilities	Package `org.codeling.utils` in the project `org.codeling.utils`
Library: IAL Meta Model	Project `org.codeling.ial.mm`
Library: Model Integration Concept	Package `org.codeling.lang.base.mic` in the project `org.codeling.core`
Library: Java Integration Mechanisms	Project `org.codeling.mechanisms`
Library: Java Transformations	Package `org.codeling.lang.base.java` in the project `org.codeling.lang.base.java`
Library: Model Transformations	Package `org.codeling.lang.base.modeltrans` in the project `org.codeling.core`
Library: Henshin	Package `org.codeling.lang.base.modeltrans.henshin` in the project `org.codeling.core`
Library: HenshinTGG	Package `org.codeling.lang.base.modeltrans.henshintgg` in the project `org.codeling.core`

Table 9.2: An overview of the code artefacts, that implement the components and libraries in Codeling

changes being propagated into the program code. The description in this section is separated into the different steps of the process. The translation process of program code into a model representation (or another implementation language during an implementation migration) is triggered via the UI after the selection of the involved languages, as described in Section 9.2.2. The UI therefor uses the transformation manager via the interface `ITransformationManager`, and executes the method corresponding to the target language, i.e `extractModelFromCode` for a translation into a specification language, or `migrateImplementation` for a translation into another implementation language.

The transformation manager component contains task definitions for each translation direction: program code to specification model, specification model to code, and implementation migration. Each of the process steps is also implemented as task, so that the more coarse grained tasks can reuse the sub tasks of the single steps. All these tasks extend Eclipse's *Jobs* framework, which makes them executable in the background of a running Eclipse platform.

In the running example the method `extractModelFromCode` of the transformation manager is called for translating the program code into a specification model. The UI provides the transformation manager with the source and the target language, and the paths to the projects in the IDE, that contain the program code to be translated. The transformation manager starts the `ModelExtractionTask`, which subsequently starts all necessary tasks for each step.

Step 1.1 – Program Code to Implementation Model

Step 1.1 is implemented in the `ProgramCodeToImplementationModelTask` type. When executed, this task first creates an empty `IDRegistry`. The ID registry is part of the *Utilities* library, because it is used throughout the whole application and in individual language plugins. This step makes use of generic implementation language definitions, which are abstract types for accessing individual translation logic. Figure 9.12 accompanies this description as an

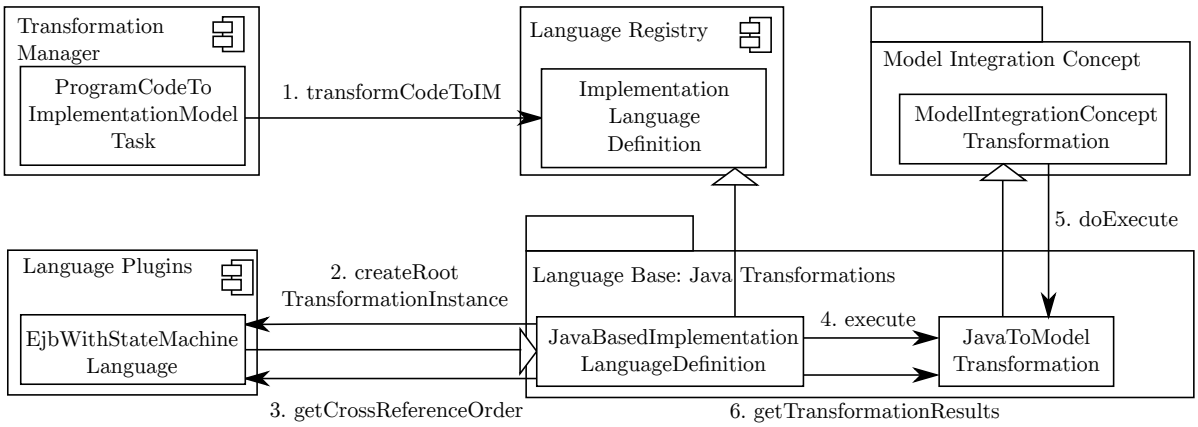


Figure 9.12: An overview of the messages between the different part of the Codeling implementation during step 1.1 and their order

overview. It shows the messages between the parts of the Codeling implementation during this step, and their order. These role of these parts and the messages are described in this section. To actually execute the translation, the corresponding method of the implementation language definition is called by the task. A framework for bidirectional Java-to-Model transformations has been developed in the context of Codeling, to support the systematic development of such translations. The framework builds upon a library with an abstract type that acts as interface towards the model integration concepts for Codeling, and a library of integration mechanisms.

Implementation Language Definitions The `LanguageDefinition` is an abstract type of the language registry. It is the root type for all definitions of languages to be used in Codeling. It declares a human readable name, a unique symbolic name, and a version of the defined language. Figure 9.13 shows the type hierarchy for language definitions in the context of the implementation language in the running example. Its types will be described in the remainder of this section. When a method is shown twice in this figure, it is declared in the supertype, and implemented in the subtype. The `ImplementationLanguageDefinition` declares abstract methods for executing the steps 1.1 to 1.3 and 6.1 to 6.3 of the Explicitly Integrated Architecture Process. Implementation language definitions need to implement these steps. The translation of step 1.1 is declared in the method `transformCodeToIM` of the type `ImplementationLanguageDefinition`. This method is called by the `ProgramCodeToImplementationModelTask`, and implemented by the specific implementation language definition type.

Java Integration Mechanisms Library The Java Integration Mechanisms library contains definitions of an excerpt of integration mechanisms, implemented with the Xtend programming language[1]. Mechanisms are Xtend type declarations, which are compatible with Java type declarations. Besides listing existing mechanisms, these types declare methods for generating

[1]Xtend – `http://www.eclipse.org/xtend/`

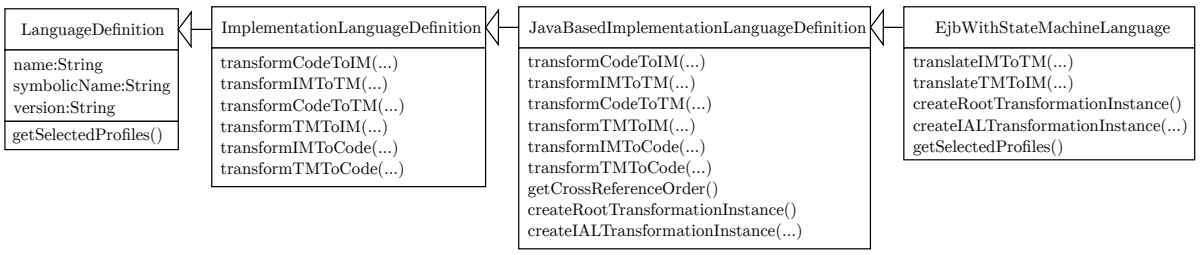


Figure 9.13: The type hierarchy of implementation language definitions

code based on the mechanism, and provide methods for evaluating the compatibility of the use of mechanisms in a meta model, both for the code generation tool (see Section 9.3). In the context of Codeling, these mechanism definitions can be used to find out whether a meta model element is translated using a specific mechanism. This is needed e.g. to find out whether a member reference of an Annotated Member Reference notation needs to be declared as an interface (when the targeted class is represented with the Marker Interface mechanism) or a type (when the targeted class is represented with the Type Annotation mechanism).

Figure 9.14 gives an overview of the type declarations for integration mechanism definitions in the context of Codeling. The abstract type `Mechanism` is the root of all mechanism declarations. It declares a method `getName` that returns the mechanism name. The method `canHandle` returns whether the given Java element is an instance of a meta model element, that is translated with the integration mechanism. The abstract types `ClassMechanism`, `AttributeMechanism`, `ReferenceMechanism`, and `ContainmentMechanism` are the basis for mechanisms of the respective abstract syntax elements of language meta models. They declare Ecore elements for the abstract syntax elements they represent.

The specific mechanism types implement the abstract methods. Listing 9.12 shows an excerpt of the Type Annotation mechanism definition type for Codeling. It extends the type `ClassMechanism`, because the Type Annotation is a class mechanism. The method `getName` returns the mechanism's name. The method `canHandle` first ensures that the code element represents a type declaration. Then it validates that the declared type has an annotation attached to it, which has the name of the meta model element. If both is true, the code element represents the model element with the Type Annotation mechanism.

Framework for Bidirectional Java-to-Model Transformations A framework for implementing the translations between Java-based program code and languages with Ecore-based meta models has been implemented for Codeling. This framework is located in the library *Language Base: Java Transformations*. The type `JavaBasedImplementationLanguageDefinition` extends the `ImplementationLanguageDefinition` and implements the necessary interaction with the framework, including the method `transformCodeToIM` for step 1.1, so that specific architecture implementation language definitions only need to configure the execution.

Listing 9.12 shows an excerpt of the implementation language definition of the running example. In the constructor a mapping between integration mechanisms and meta model classes is created. This mapping can be used during the transformation, e.g. during the translation of a model notation, that follows the Annotated Member Reference mechanism. The program

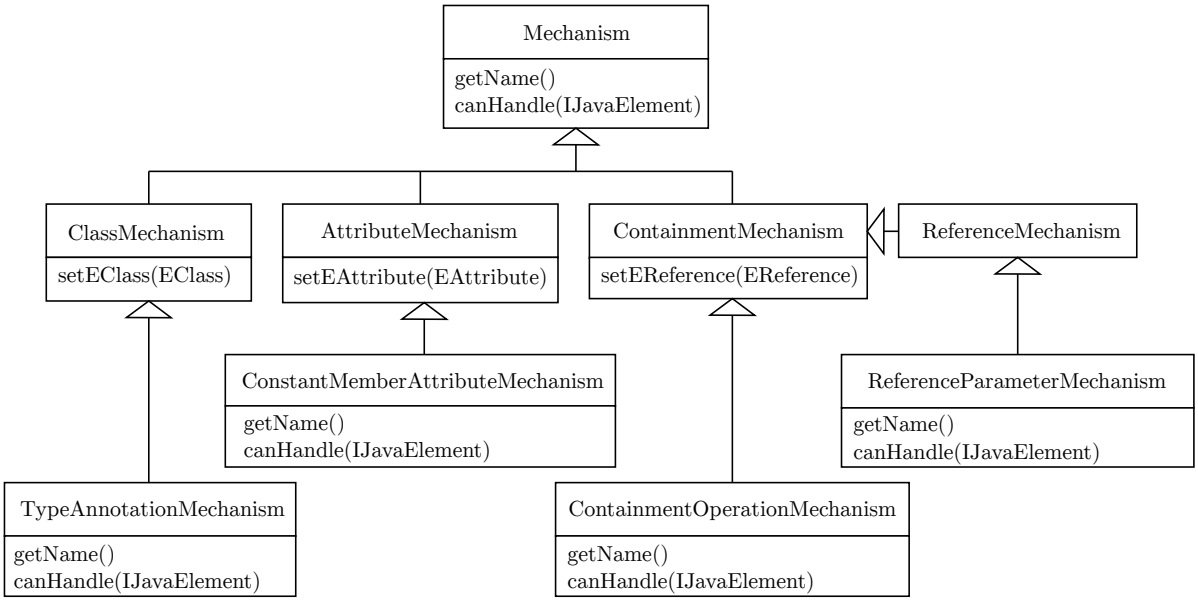


Figure 9.14: The type hierarchy of integration mechanisms in the Java Integration Mechanisms library with some specific mechanisms

```
class TypeAnnotationMechanism extends ClassMechanism {
  [...]
  override getName() {
    "Type␣Annotation"
  }
  override canHandle(IJavaElement codeElement, ENamedElement metaModelElement) {
    return codeElement instanceof IType &&
      (codeElement as IType).getAnnotation(metaModelElement.name).exists;
  }
}
```

Listing 9.12: An excerpt of the Type Annotation mechanism definition type for Codeling, implemented with the Xtend programming language

code in the model notation of this mechanism depends on the mechanism, that is used for the reference's target class. Finally the constructor declares an order in which non-containment references are translated. In the example, a specific ordering is not necessary. The method `createRootTransformationInstance` creates an instance of a transformation type for the root of the architecture implementation language (message number 2 in Figure 9.12). The cross reference order is made available via the method `getCrossReferenceOrder` (message number 3 in Figure 9.12).

```
public class EjbWithStatemachineLanguage extends                                    1
    JavaBasedImplementationLanguageDefinition {
                                                                                     2
  public EjbWithStatemachineLanguage() {                                             3
    ejbWithSMPackage.eINSTANCE.getName(); // Initialize the Ecore package            4
                                                                                     5
    // Create the mechanism mapping                                                  6
    MechanismsMapping m = MechanismsMapping.getInstance();                           7
    ejbWithSMPackage i = ejbWithSMPackage.eINSTANCE;                                 8
    m.put(i.getStateMachine(), TypeAnnotationMechanism.class);                       9
    m.put(i.getState(), MarkerInterfaceMechanism.class);                             10
    m.put(i.getStateMachine_States(), AnnotatedMemberReferenceMechanism.class);      11
    m.put(i.getStateMachine_Initial(), AnnotatedMemberReferenceMechanism.class);     12
    m.put(i.getState_Transition(), ContainmentOperationMechanism.class);             13
    m.put(i.getTransition_Target(), NinjaSingletonContainmentReferenceMechanism.class);14
                                                                                     15
    [...]                                                                            16
                                                                                     17
    // Add all EClasses to the cross references. The order is irrelevant            18
    for (final EClassifier c : ejbWithSMPackage.eINSTANCE.getEClassifiers())         19
      if (c instanceof EClass)                                                       20
        crossReferenceOrder.add((EClass) c);                                         21
                                                                                     22
  }                                                                                  23
                                                                                     24
  @Override                                                                          25
  public AbstractModelCodeTransformation<? extends EObject, ? extends IJavaElement> 26
      createRootTransformationInstance() {
    return new ArchitectureTransformation();                                         27
  }                                                                                  28
                                                                                     29
  [...]                                                                              30
}                                                                                    31
```

Listing 9.13: An excerpt of the implementation language definition type for the running example, that configures the bidirectional Java to Ecore transformations

The now configured translation of step 1.1 is triggered by the `JavaBasedImplementationLanguageDefinition`. It executes a Java code to model transformation (messages number 4 and 5 in Figure 9.12), which it configures using the program code's projects, the ID registry, a root transformation instance, and a non-containment reference order as defined by the specific implementation language definition. General structures and types for bidirectional model-code transformations based on the Model Integration Concept are implemented in the *Model Integration Concept* library. The transformation results are collected after the transformation finished (message number 6 in Figure 9.12).

Model Integration Concept Library The Model Integration Concept library contains an abstract type `ModelIntegrationConceptTransformation` for defining bidirectional model-code transformations. Such a transformation takes a list of project paths and an `IDRegistry` as input. Once successfully executed, it returns a `TransformationResult`, which includes the resulting model and the updated `IDRegistry`. The type `TransformationResult` is contained in the *Utilities* library, because it is used throughout the whole application and within the individual language plugins.

Java Transformations Library The Java Transformations library is the basis for bidirectional code transformations between Java code and architecture implementation language models. The type `JavaToModelTransformation` extends the abstract type `ModelIntegrationConceptTransformation` of the Model Integration Concept library and executes the actual transformations. The transformations between Java code and Ecore models in the context of Codeling is designed as follows: For each meta model class, attribute, and reference, a transformation type has to be implemented. The root type of all such transformations is the type `AbstractModelCodeTransformation`. This type extends `java.util.concurrent.RecursiveAction`. It is therefore prepared to be a concurrently running task in Java's `java.util.concurrent.ForkJoinPool`. The `ForkJoinPool` is a pool of worker threads that execute tasks. In the default configuration, the number of threads, that are concurrently executed by the pool, is always less or equal to the number of processing cores on the executing machine. Worker threads within the pool take one task at a time. These tasks are allowed to create further tasks in the pool, building a tree of tasks to be performed. This structure allows the subtypes of `AbstractModelCodeTransformation` to be executed concurrently and to spawn child transformations.

Figure 9.15 gives an overview of the type hierarchy of transformations in this library, focused on the steps 1.1 to 1.3. The types, attributes, and methods shown in this figure are described in the respective steps. Codeling distinguishes between class, attribute, and reference transformations based on integration mechanisms. The abstract types `ClassMechanismTransformation`, `AttributeMechanismTransformation`, and `ReferenceMechanismTransformations` are the basis for model notation transformations for their types of meta model elements. They store a reference to the meta model element to be translated.

A transformation of code into a model representation is executed as follows: A root transformation object first translates the root code element—usually the projects at the given paths—into a model representation, the root node of the targeted model, using the method `transformToModel`. The transformation objects stores references to a primary code element (e.g. a type declaration for the Type Annotation mechanism) and the corresponding model element, effectively creating a mapping between the code and the model.

After the translation, the transformation object is added to a transformation object registry called `FindTranslatedElements`. This registry can be used later to retrieve model elements, that represent specific code elements or vice versa. At last, the transformation creates child transformation objects for its attributes and containment references using the method `createChildTransformationsToModel` and adds them to the pool of tasks.

Transformations for classes have transformations for their attributes and containment references as child transformations. Transformations for attributes have no child transformations. Reference transformations, including containment reference transformations, have transformations for their target objects as child references. The `ReferenceMechanismTransformations`

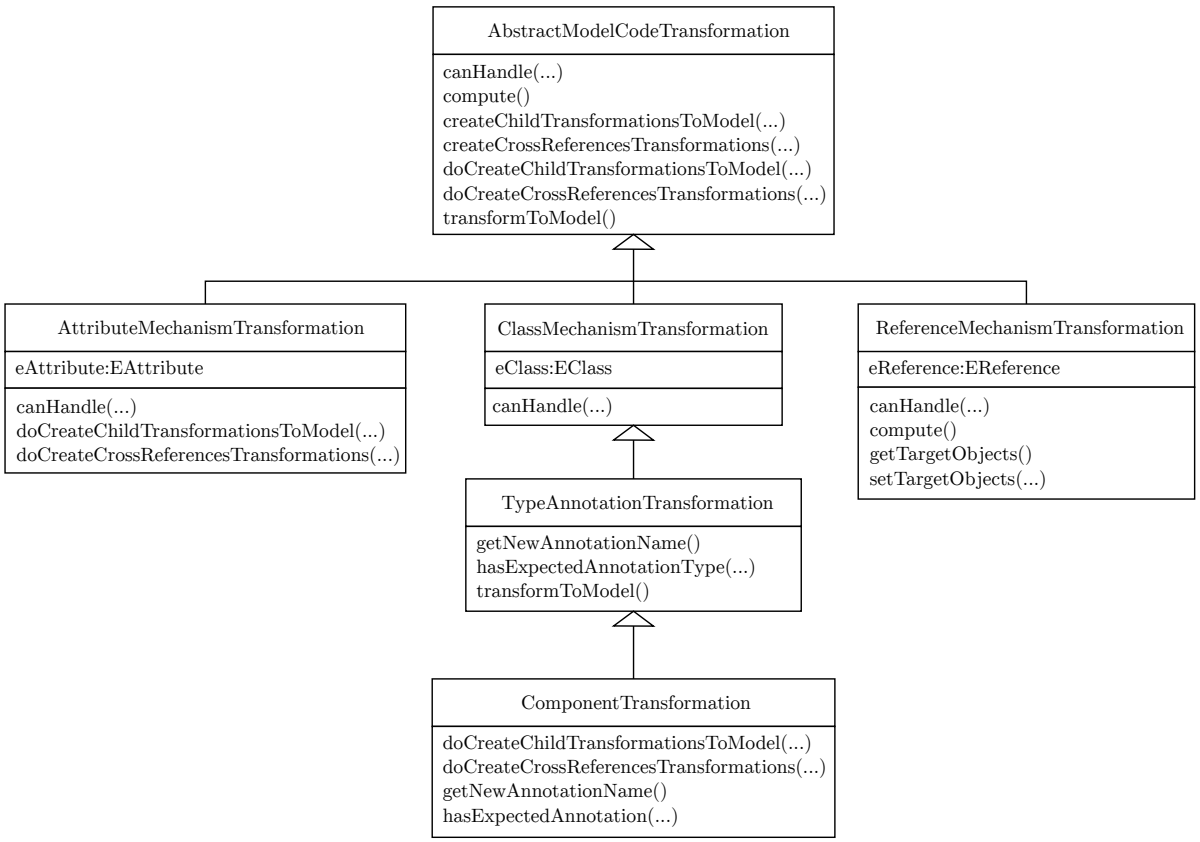


Figure 9.15: Excerpt of the type hierarchy of transformations in the bidirectional model-code transformation framework, with attributes and methods for the steps 1.1 and 1.3

changes the execution order for transformations. If a containment reference is translated from a code to a model representation, the targets of the reference do not exist, because they have not been translated yet. Therefore the `ReferenceMechanismTransformations` first submits the translation for the target objects to the thread pool and awaits their results. Then the reference notation can be translated.

Listing 9.14 shows the `ComponentTransformation` of the running example. It extends the `TypeAnnotationTransformation` for the *Component* class in the example meta model. The constructor stores the parent transformation and specifies the *Component* class as translated meta model element. The method `doCreateChildTransformationsToModel` add instances of child transformations to a result list, for the attributes (none in the example) and containment references owned by the *Component* class. The method `doCreateCrossReferencesTransformations` adds an instance of the transformation for the non-containment reference called *references*. The methods `hasExpectedAnnotation` and `getNewAnnotationName` configure the supertype `TypeAnnotationTransformation`, which is shown in Listing 9.15. The notation for the *Component* does not exactly comply to the Type Annotation mechanism, because the annotation for the type is not called `Component`, but the notation relies on the existing API for EJB. The expected annotation can be changed using these methods. The method `hasExpectedAnnotation` evaluates whether they given

type declaration has one of the EJB session bean annotations attached. The method `getNewAnnotationName` returns an annotation for newly created components in code.

```
public class ComponentTransformation extends TypeAnnotationTransformation<Component> {
  public ComponentTransformation(
      AbstractModelCodeTransformation<? extends EObject, ? extends IJavaElement>
          parentTransformation) {
    super(parentTransformation, ejbWithSMPackage.eINSTANCE.getComponent());
  }

  @Override
  public void doCreateCrossReferencesTransformations(
      List<AbstractModelCodeTransformation<? extends EObject, ? extends IJavaElement>>
          result) {
    result.add(new ReferencesTransformation(this));
  }

  @Override
  protected void doCreateChildTransformationsToModel(
      List<AbstractModelCodeTransformation<? extends EObject, ? extends IJavaElement>>
          result) {
    result.add(new StatemachineTransformation(this));
    result.add(new OperationsTransformation(this));
  }

  @Override
  public boolean hasExpectedAnnotation(IType type) {
    List<String> ls = Arrays.asList("Stateful", "javax.ejb.Stateful", "Stateless", "
        javax.ejb.Stateless",
        "Singleton", "javax.ejb.Singleton");
    for (String s : ls) {
      if (type.getAnnotation(s).exists())
        return true;
    }
    return false;
  }

  @Override
  protected String getNewAnnotationName() {
    return "javax.ejb.Stateful";
  }
}
```

Listing 9.14: The bidirectional Java-to-Model transformation for the *Component* class in the meta model of the running example

Listing 9.15 shows an excerpt of the generic transformation type for the Type Annotation mechanism. In the method `transformToModel` it creates a model element and sets the name attribute according to the definition of the Type Annotation mechanism. The excerpt also shows the default implementation of the method `hasExpectedAnnotation`. The name of the class is used as annotation name. This annotation must be created in a corresponding library for the result to compile.

After translating all class, attribute, and containment notations, all objects of the target model exist. Now the transformation framework revisits all class transformation objects. For each of these transformations, translations for non-containment references are created and invoked, following the ordering given by the specific implementation language definition. After the code-to-model transformations, all translated code and model elements are registered in the ID registry. The results are wrapped in a `TransformationResult` object, which comprises the implementation model and the ID registry.

```
abstract class TypeAnnotationTransformation<ELEMENTECLASS extends EObject> extends
    ClassMechanismTransformation<ELEMENTECLASS, IType> {

  override transformToModel() throws CodelingException {
    // Create element
    modelElement = eClass.EPackage.EFactoryInstance.create(eClass);

    // Set name attribute value
    val String name = codeElement.elementName;
    setNameAttributeValue(modelElement, name);

    return modelElement;
  }

  public def boolean hasExpectedAnnotation(IType type) {
    return type.getAnnotation(
      eClass.name.toFirstUpper
    ).exists;
  }

  [...]

}
```

Listing 9.15: An excerpt of the abstract bidirectional Java-to-Model transformation for the Type Annotation mechanism, implemented with the Xtend programming language

Step 1.2 – Implementation Model to Translation Model

The task for step 1.2 of the process uses the method `transformIMToTM` of the type `ImplementationLanguageDefinition` to execute the model-to-model transformation. The *Model Transformations* library contains an API for executing model transformations in the context of Codeling in the form of an abstract type `ModelTransformation`. The language definitions developed in this thesis implement the transformations of this step using HenshinTGG. Figure 9.16 gives an overview of the messages during the execution of this task.

To execute these transformations, a utility type named `HenshinTGGBasedLanguageDefinitionHelper` exists, which implements this recurring logic. In this step, first a model is translated into a translation model representation using HenshinTGG. Then, the newly created translation model elements are added to the ID registry. Thereby a mapping is created between program code structures, the implementation model element they represent, and the translation model element they represent. Listing 9.16 shows an excerpt of the implementation language definition of the running example, that triggers the translation from the implementation model to the translation model with the `HenshinTGGBasedLanguageDefinitionHelper`.

The HenshinTGG tool is used programmatically during this translation. The library `HenshinTGG` contains the type `HenshinTGGTransformation`, an implementation of the abstract `ModelTransformation` type, for executing HenshinTGG transformations within Codeling. It includes the execution of TGG transformations using the HenshinTGG engine, and the propagation of IDs between the ID registry and the models. As no generic API for HenshinTGG existed at the time of developing Codeling, such a generic API for HenshinTGG has been developed in this context, so that HenshinTGG can be used without using its user interface.

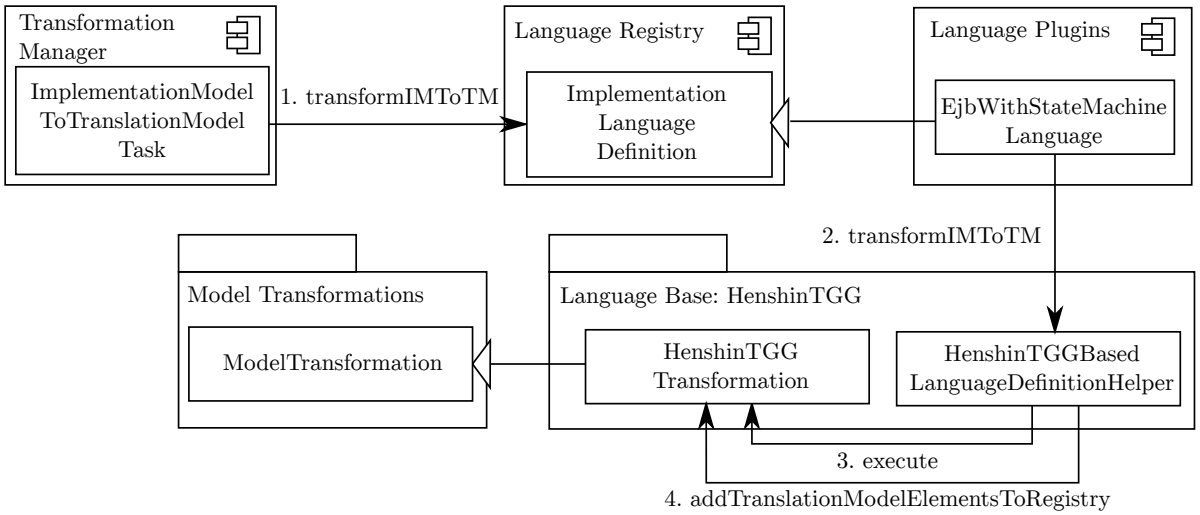


Figure 9.16: An overview of the messages between the different part of the Codeling implementation during step 1.2 and their order

```
public class EjbWithStateMachineLanguage extends                                    1
    JavaBasedImplementationLanguageDefinition {
                                                                                     2
  final URI henshinTGGFileURI = URI.createPlatformPluginURI(                         3
      "/" + FrameworkUtil.getBundle(getClass()).getSymbolicName() + "/AIL2IAL.henshin",4
          true);
                                                                                     5
  [...]                                                                              6
                                                                                     7
  @Override                                                                          8
  public TransformationResult transformIMToTM(List<EObject> cmRoots, IDRegistry      9
      idRegistry) throws CodelingException {
    return new HenshinTGGBasedLanguageDefinitionHelper().transformCMToIL(this,       10
        henshinTGGFileURI, cmRoots,
        idRegistry, monitor);                                                        11
  }                                                                                  12
                                                                                     13
  [...]                                                                              14
                                                                                     15
}                                                                                    16
```

Listing 9.16: An excerpt of the implementation language definition type for the running example, that triggers the translation from the implementation model to the translation model with HenshinTGG

The input for a HenshinTGG transformation in Codeling is an existing model to be translated, a language definition (the second language is always the Intermediate Architecture Description Language), a HenshinTGG rule file, and a direction: forward or backward. It first loads the model into the TGG engine, executes the TGG rules in the rule file, and exports the desired model from the TGG engine. The desired model is determined by the declared transformation direction. After a forward transformation the target model of the triple graph is exported. After a backward transformation the source model is exported. In step 1.2, the translation is directed from the implementation model to a translation model. Therefore the translation model is exported. The results are wrapped in a `TransformationResult` object, which comprises the translation model and the updated ID registry.

Step 1.3 – Program Code to Translation Model

The task for step 1.3 of the process uses the method `transformCodeToTM` of the type `ImplementationLanguageDefinition` to extract translation model information from the program code, which cannot be expressed in the architecture implementation language. Figure 9.17 gives an overview of the messages during the execution of this task. The framework for bidirectional Java-to-Model transformations is used for these translations. In the framework, these transformations are called *IAL transformations*, because they translate between the IAL and the program code directly.

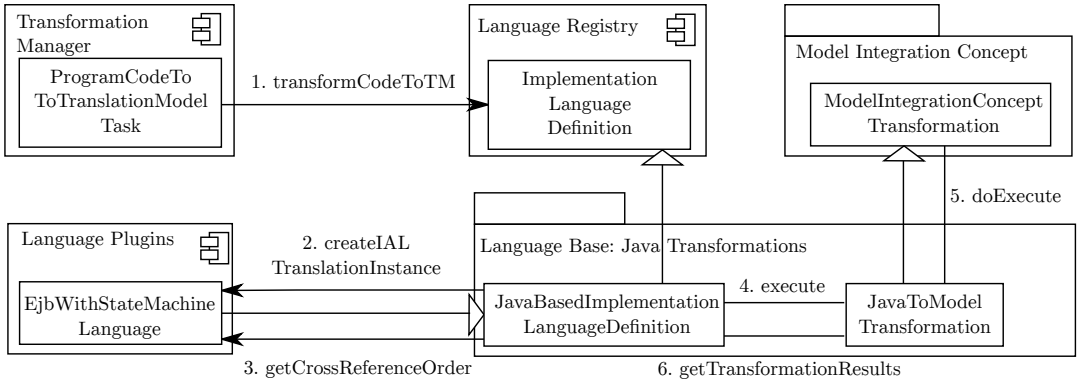


Figure 9.17: An overview of the messages between the different part of the Codeling implementation during step 1.3 and their order

Framework for Bidirectional Java-to-Model Transformations The type `JavaBasedImplementationLanguageDefinition` implements the method `transformCodeToTM` (see Listing 9.17). It takes the list of projects, an ID registry, the architecture implementation model, and the translation model as input. For each entry in the ID registry the implementation model element and the translation model element are resolved. Then for each implementation model element, IAL transformation instances are created, configured, and executed using the `ModelToJavaTransformation` type as executor, following the same process as the translation in step 1.1. Due to technical constraints of the EMF Profiles implementation, the profile applications are created in a separate file. This file is merged after

the translation, so that the resulting model needs to be aggregated into one file. At last, the results are wrapped with a `TransformationResult` object, containing the updated translation model and the updated ID registry.

```
public abstract class JavaBasedImplementationLanguageDefinition extends
    ImplementationLanguageDefinition {
  [...]

  public TransformationResult transformCodeToTM(List<IJavaProject> projects, List<
      EObject> imRoots,
      List<EObject> tmRoots, IDRegistry idRegistry) throws CodelingException {
    idRegistry.getRegistryEntries().keySet().stream().forEach(id -> {
      final EObject imElement = idRegistry.getImplementationModelElement(id, imRoots);
      if (imElement == null)
        // If this element has not implementation model representation, ignore it.
        return;

      final EObject tmElement = idRegistry.getTranslationModelElement(imElement,
          tmRoots);
      final List<? extends IALTransformation<?, ?>> ialTransformations =
          createIALTransformationInstance(imElement);
      if (ialTransformations != null)
        ialTransformations.stream().forEach(t -> {
          t.setIDRegistry(idRegistry);
          t.setModelElement(imElement);
          t.getIALHolder().setFoundationalIALElement(tmElement);

          JavaToModelTransformation executor = new JavaToModelTransformation(projects,
              idRegistry, (AbstractModelCodeTransformation<?, ?>) t, null);
          executor.execute(monitor);
        });
    });

    // Create a temporary profile application file for using the emf profiles facade.
    final Resource applicationResource = getResource(rSet,
        CodelingConfiguration.DEBUG_MODELDIR_PATH + "profile-application.pa.xmi");
    mergeStereotypeApplicationsWithILModel(tmRoots, applicationResource);
    return new TransformationResult(tmRoots, idRegistry);
  }

  [...]
}
```

Listing 9.17: An excerpt of the generic implementation for translating program code to translation model elements. Exception handling has been removed for readability reasons.

IAL Transformations IAL transformations in the framework are subtypes of the `AbstractModelCodeTransformation`, that additionally implement an interface `IALTransformation`. The interface defines three methods. The method `getIALHolder` returns an object that holds references to IAL model and code elements. For this step, the holder references the current foundational translation model element – i.e. the IAL model element, that represents the implementation model element, and the code element that represents the translation model element, i.e the result of this step. The method `resolveTranslatedIALElement` returns the translation model element, which the IAL transformation type translates. the method `codeFragmentExists` returns whether the code fragment for the transformation model element is attached to the underlying program code. These methods are used to configure the IAL transformations, so that the translation model element and code element are available during the transformation.

The implementation language definition declares which transformation types are to be instantiated. When the translation model contains information, that the implementation language cannot express, this information is always attached to an implementation model element. In the running example, the parent-child relationship between components should be represented. The implementation model class, that this information should be attached to, is the *Component*. This class is translated to a *ComponentType* of the IAL in step 1.2. In the translation model a parent-child relationship between two components is declared in the profile *Scoped Component Hierarchy* (see Definition 87). The profile contains a stereotype *HierarchicalComponentTypeScoped*, that can be applied to component types. It contains a reference *childTypes* towards component types. Listing 9.18 shows an excerpt of the language definition, where this aspect is declared. In the constructor of the language definition type, a map is filled. The map's key is the implementation model element. The value is a tuple that contains the IAL abstract syntax element (the reference *childTypes* in the example) and a list of transformation types.

```
public class EjbWithStatemachineLanguage extends                                    1
    JavaBasedImplementationLanguageDefinition {
                                                                                     2
  public EjbWithStatemachineLanguage() {                                             3
    [...]                                                                            4
                                                                                     5
    // Create the map of model elements to reference transformations                 6
    ialTransformations.put(i.getComponent(),  new IALTransformationTuple(            7
      ProfilesUtils.getEReference(Profiles.COMPONENTS_HIERARCHY_SCOPED.load(),       8
        "HierarchicalComponentTypeScoped", "childTypes"),                            9
      Arrays.asList(ChildTransformation.class)));                                    10
                                                                                     11
    [...]                                                                            12
  }                                                                                  13
                                                                                     14
  [...]                                                                              15
}                                                                                    16
```

Listing 9.18: An excerpt of the implementation language definition type for the running example, that configures the bidirectional Java to Ecore transformations

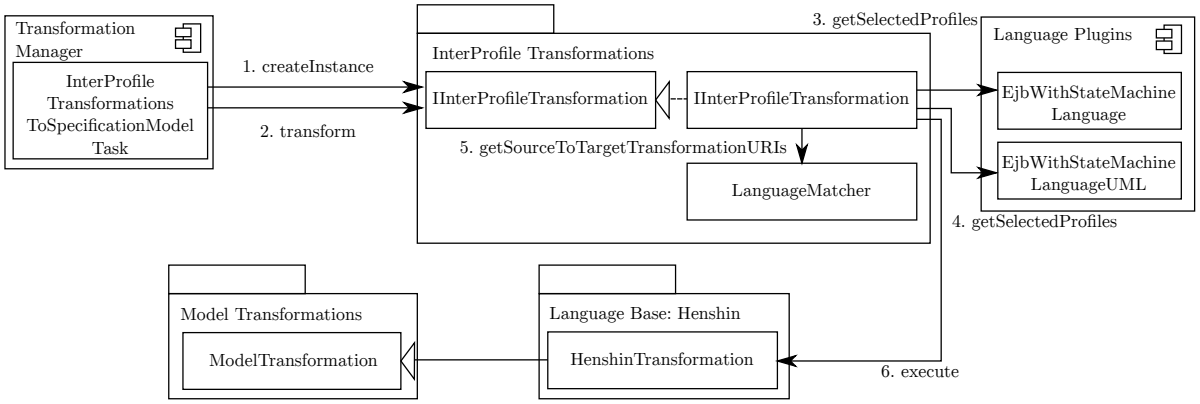


Figure 9.18: An overview of the messages between the different part of the Codeling implementation during step 2 and their order

Step 2 – Inter-Profile Transformations

For executing step 2 of the process, the task `InterProfileTransformationToImplementationModelTask` uses the component *Inter-Profile Transformations* (see Figure 9.18). The interface provides a method `transform`, which takes the source and the target language as input, as well as a source model, and an ID registry object. The component first requests the profiles used by the languages by using the corresponding method `getSelectedProfiles` of the language definitions. The language matcher within the component then determines, which inter-profile transformations and profile activation rules need to be executed. The corresponding rule files are then executed sequentially. When languages define architecture model transformations using HenshinTGG, the profiles used by the language are declared within the HenshinTGG rule file. The profiles need to be imported in the rule file for making their elements available to the rules. The type `HenshinTGGBasedLanguageDefinitionHelper` provides a default implementation, which extracts the profiles from the rule file. Languages that define architecture model transformations using HenshinTGG therefore do not need to explicitly implement this method.

The translation is executed by the type `HenshinTransformation` of the *Henshin* library. It is an implementation of the abstract type `ModelTransformation`, which has been described in step 1.2. For the translation, it takes a rule file URI and a translation model as input. By convention the rules must always have a unit or rule named `main`, which is executed by the `HenshinTransformation` type. After the transformation, the results are wrapped in a `TransformationResult` object, which comprises the updated translation model, and an unchanged ID registry.

Step 3 – Translation Model to Specification Model

The task for step 3 of the process uses the method `transformToSM` of the type `SpecificationLanguageDefinition` to execute the model-to-model transformation from the translation model to the specification model (see Figure 9.19). This step reuses the

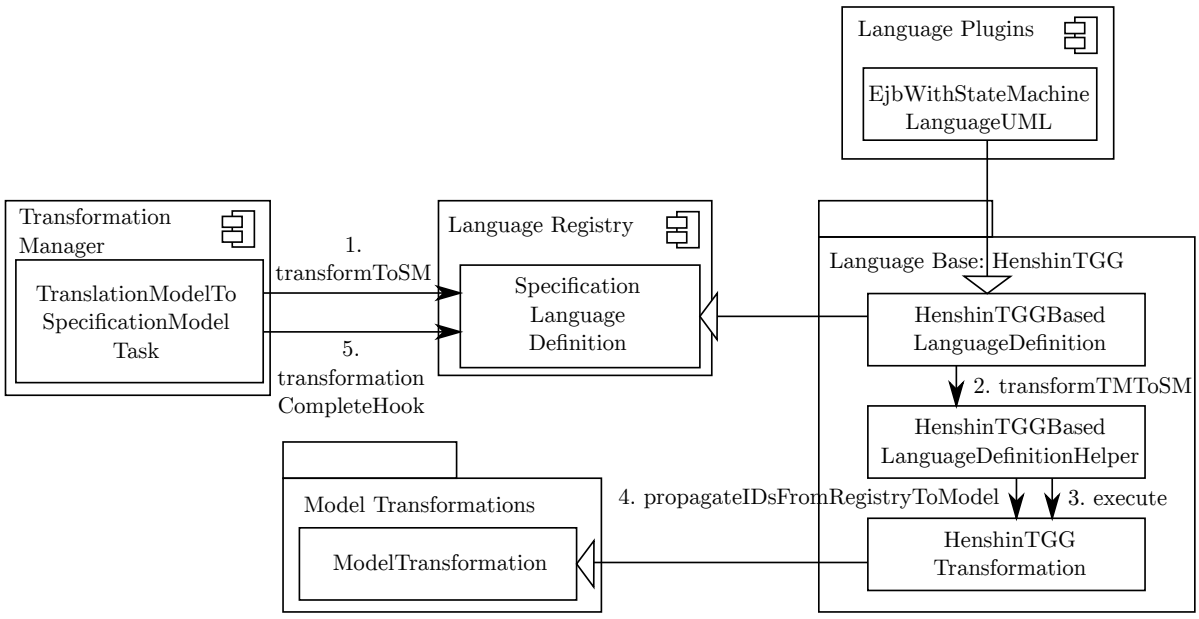


Figure 9.19: An overview of the messages between the different part of the Codeling implementation during step 3 and their order

implementation for HenshinTGG transformations, that have already been described in step 1.2. In this step of the running example, the architecture specification language UML is targeted. To make this language available in Codeling, a `SpecificationLanguageDefinition` has to be implemented. Figure 9.20 shows the type hierarchy for specification language definitions in the running example. The `SpecificationLanguageDefinition` declares abstract methods for executing the steps 3 and 4 of the Explicitly Integrated Architecture Process. Specification language definitions need to implement these steps.

The translation of step 3 is declared in the method `transformToSM` of the type `SpecificationLanguageDefinition`. Listing 9.19 shows the specification language definition for the UML as defined in the running example. It extends the type `HenshinTGGBasedLanguageDefinition`, which contains the default implementation for architecture model transformations with HenshinTGG. The specific language definition declares a HenshinTGG rule file, which is used by the super type. The UML meta model is part of the Eclipse Modeling Framework. It does not declare an identifying attribute for all elements. Thus the methods `setID` and `getID` have to be overridden. In the running example, structured comments are used to attach an ID to UML model elements. The type `SpecificationLanguageDefinition` also declares a method `transformationCompleteHook`, which is automatically called by the task during this step, after the transformation is completed. This method can be implemented by a specific specification language definition, e.g. to create diagrams from the resulting specification model or to split the specification model into multiple files, as it might be required for its editor.

```
public class EjbWithStateMachineLanguageUML extends HenshinTGGBasedLanguageDefinition {
                                                                                        2
  final static URI henshinTGGFileURI = URI.createPlatformPluginURI(                     3
      "/" + FrameworkUtil.getBundle(EjbWithStateMachineLanguageUML.class).              4
          getSymbolicName() + "/IAL2UML.henshin",
      true);                                                                            5
                                                                                        6
  public EjbWithStateMachineLanguageUML() {                                             7
    super(henshinTGGFileURI);                                                           8
  }                                                                                     9
                                                                                        10
  /**                                                                                   11
   * Creates a comment in the UML model to save the element's id in Codeling           12
   */                                                                                   13
  @Override                                                                             14
  public void setID(EObject object, String id) {                                       15
    Comment comment = UMLFactory.eINSTANCE.createComment();                             16
    comment.setBody("Codeling-ID:␣" + id);                                              17
    ((Element) object).getOwnedComments().add(comment);                                 18
  }                                                                                     19
                                                                                        20
  /**                                                                                   21
   * Retrieves the id from a comment in the UML model as it was stored by setID.       22
   */                                                                                   23
  @Override                                                                             24
  public String getID(EObject object) {                                                25
    for (Comment c : ((Element) object).getOwnedComments())                             26
      if (c.getBody().matches("Codeling-ID:␣.+"))                                       27
        return c.getBody().substring("Codeling-ID:␣".length());                         28
    return null;                                                                        29
  }                                                                                     30
}                                                                                       31
```

Listing 9.19: The transformation of a translation model to a specification model via HenshinTGG

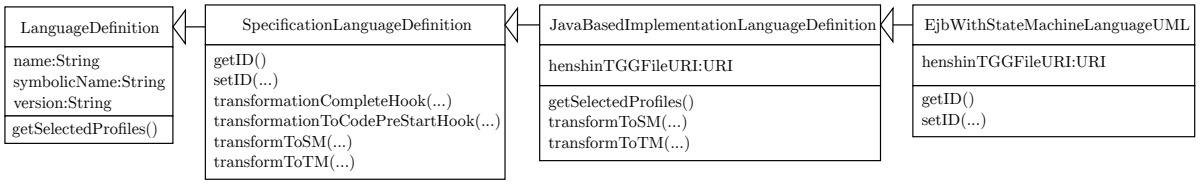


Figure 9.20: The type hierarchy of specification language definitions

The super type delegates the actual transformation to the method `transformTMtoSM` of the type `HenshinTGGBasedLanugageDefinitionHelper`. The trigger of this process is shown in Listing 9.20. First, the HenshinTGG API of Codeling is used to execute a forward translation as defined by the HenshinTGG file for the specification language. Then the IDs of the newly created translation model elements are added to the corresponding ID registry elements. The result of step 3 is a file named `specification-model.xmi` in a project named `architecture-carrying-code-temp`, which contains the specification model.

```
public class HenshinTGGBasedLanguageDefinitionHelper {
  [...]
  public TransformationResult transformTMToSM(HenshinTGGBasedLanguageDefinition
      henshinTGGBasedLanguageDefinition,
      URI henshinTGGFileURI, List<EObject> tmRoots, IDRegistry idRegistry,
          IProgressMonitor monitor) {
    final HenshinTGGTransformation task = new HenshinTGGTransformation(
        henshinTGGBasedLanguageDefinition,
        henshinTGGFileURI, "Transforming␣Translation␣Model␣to␣Specification␣Model",
            TGGDirection.FORWARD, idRegistry, tmRoots);
    TransformationResult result = task.execute(monitor);
    task.propagateIDsFromRegistryToModel();
    return result;
  }
  [...]
}
```

Listing 9.20: The transformation of a translation model to a specification model via HenshinTGG

The model is now ready to be changed in an editor. In the running example an Ecore-based UML model has been created. An example for an editor that can handle UML models specified with the UML meta model for Ecore is Papyrus[2]. After changing the model as described in the use case in Section 9.2.1, the changed model is translated to the code using the task `IntegrateModelTask`, which executes the following steps.

Step 4 – Specification Model to Translation Model

The task for step 4 of the process uses the method `transformToTM` of the type `SpecificationLanguageDefinition` to execute the model-to-model transformation from the specification model to the translation model. This step also reuses the implementation for HenshinTGG transformations, that have already been described in step 1.2. Figure 9.21 gives an overview of the messages during the execution of this task.

In step 3, the method `transformationCompleteHook`, was executed automatically after the transformation. In this step, the counterpart `transformationToCodePreStartHook` is called before the translation. This can be used e.g. to aggregate a model that has been edited in multiple files, which have been created from the specification model after step 3.

In this step the changes in the specification model are propagated to the translation model. First the translation model and the ID registry resulting from step 2 is loaded. They represent

[2]Papyrus – A UML Editor based on Eclipse – `https://www.eclipse.org/papyrus/`

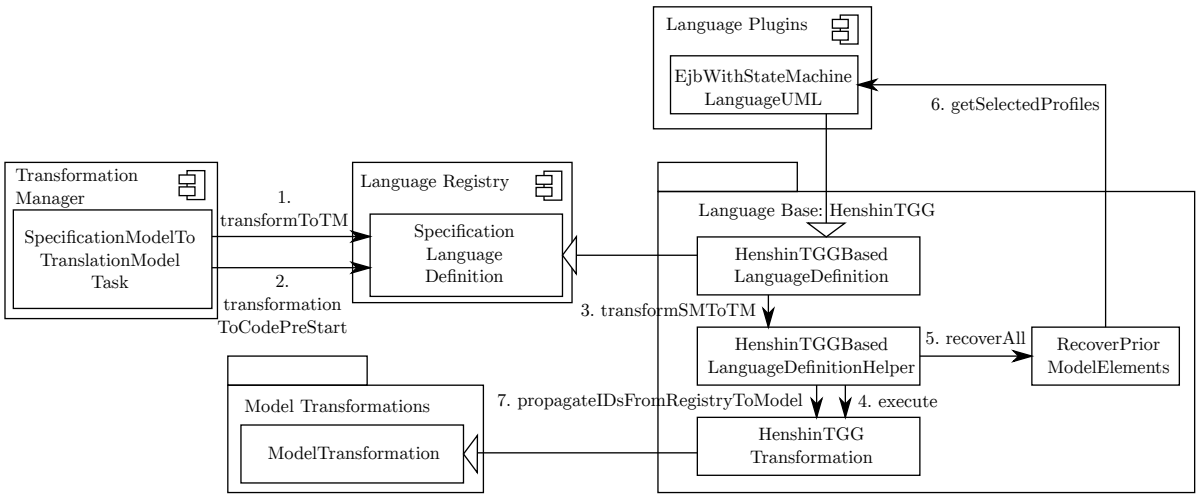


Figure 9.21: An overview of the messages between the different part of the Codeling implementation during step 4 and their order

the system's architecture before the change. These and the changed specification model are used as input for the method `transformToTM` of the type `SpecificationLanguageDefinition`. The actual translation is forwarded to the type `HenshinTGGBasedLanguageDefinitionHelper`, which implements this logic (see Listing 9.21). The transformation file, the changed specification model, the original translation model, and the original ID registry are used as input. First, a backwards translation is executed. This translates the specification model into a translation model. This translation model only contains model elements of profiles, that the specification language uses. In the running example, the architecture implementation language describes a flat component hierarchy, while the specification language describes a scoped component hierarchy. The profile application *HierarchicalArchitectureFlat* is not translated to the translation model, because the specification language does not use that profile, and therefore the specification model does not have this information. These elements and all their attributes and references are recovered from the original, unchanged translation model, which was the result of step 2, using the type `RecoverPriorModelElements`. At last the IDs of all newly created elements are added to the ID registry, and all ID registry entries of deleted elements are removed. The result of step 4 is wrapped in a `TransformationResult` object, which contains the translation model and the updated ID registry.

```
public class HenshinTGGBasedLanguageDefinitionHelper {

  [...]

  public TransformationResult transformSMToTM(HenshinTGGBasedLanguageDefinition
      henshinTGGBasedLanguageDefinition,
      URI henshinTGGFileURI, List<EObject> tmRoots, List<EObject> preChangeTMRoots,
          IDRegistry idRegistry,
      IProgressMonitor monitor) {

    // Execute =Backwards Propagation=
    final HenshinTGGTransformation bwppgTask = new HenshinTGGTransformation(
        henshinTGGBasedLanguageDefinition,
        henshinTGGFileURI, "Executing␣Backwards␣Propagation␣Rules␣for␣Specification␣
            Model␣to␣Translation␣Model",
        TGGDirection.BACKWARD_PROPAGATION, idRegistry, tmRoots);
    final TransformationResult result = bwppgTask.execute(monitor);

    // Recover elements lost during the translation
    Models.store(result.getModelRoots(), CodelingConfiguration.DEBUG_MODELDIR_PATH + "
        tmp-recover-TM.xmi");
    new RecoverPriorModelElements().recoverAll(preChangeTMRoots, result.getModelRoots()
        ,
        henshinTGGBasedLanguageDefinition, result.getIdRegistry(), idRegistry);

    // Update the id registry
    List<String> updatedIDs = bwppgTask.propagateIDsFromModelToRegistry();
    Set<String> deletedIDs = idRegistry.getAllIds();
    deletedIDs.removeAll(updatedIDs);
    idRegistry.deleteEntries(deletedIDs);

    return result;
  }

  [...]
}
```

Listing 9.21: The propagation changes in the specification model to the transformation model via HenshinTGG

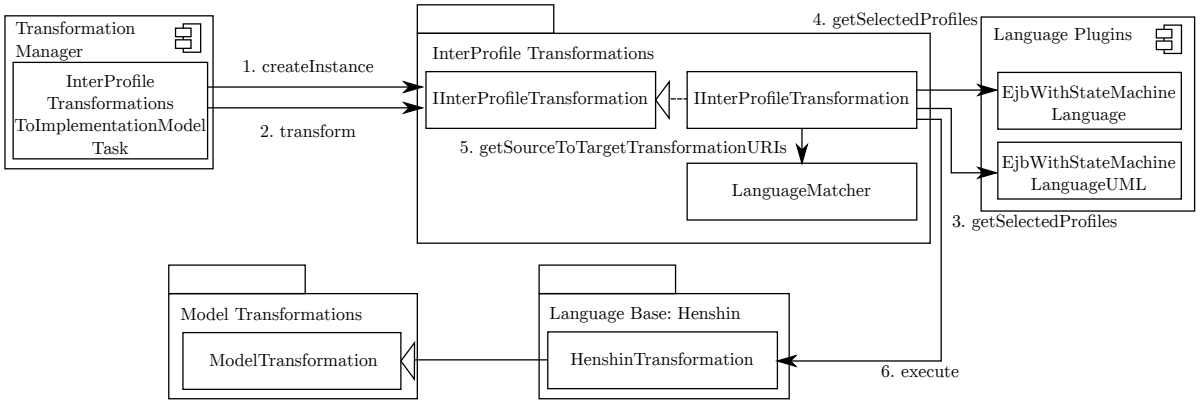


Figure 9.22: An overview of the messages between the different part of the Codeling implementation during step 5 and their order

Step 5 – Inter-Profile Transformations

For executing step 5 of the process, the task `InterProfileTransformationToSpecificationModelTask` uses the component *Inter-Profile Transformations*. This step works analogously to step 2 (Inter-Profile Transformations in the code-to-model direction), but the source and target languages are swapped. Figure 9.22 gives an overview of the messages during the execution of this task. After the transformation, the results are wrapped in a `TransformationResult` object, which comprises the updated translation model, and an unchanged ID registry.

Step 6.1 – Translation Model to Implementation Model

The task for step 6.1 of the process uses the method `transformIMToTM` of the type `ImplementationLanguageDefinition` to execute the model-to-model transformation. The type `TranslationModelToImplementationModelTask` implements this step. Figure 9.23 gives an overview of the messages during the execution of this task.

Step 6.1 works analogously to step 1.2 (Implementation Model to Translation Model), but the translation is executed in the opposite direction. Listing 9.22 shows an excerpt of the implementation language definition of the running example, that triggers the translation from the implementation model to the translation model with the `HenshinTGGBasedLanguageDefinitionHelper`. The helper executes a TGG transformation from the translation model to the implementation model, and propagates updated paths of implementation model elements in the ID registry. The results are wrapped in a `TransformationResult` object, which comprises the implementation model and the updated ID registry.

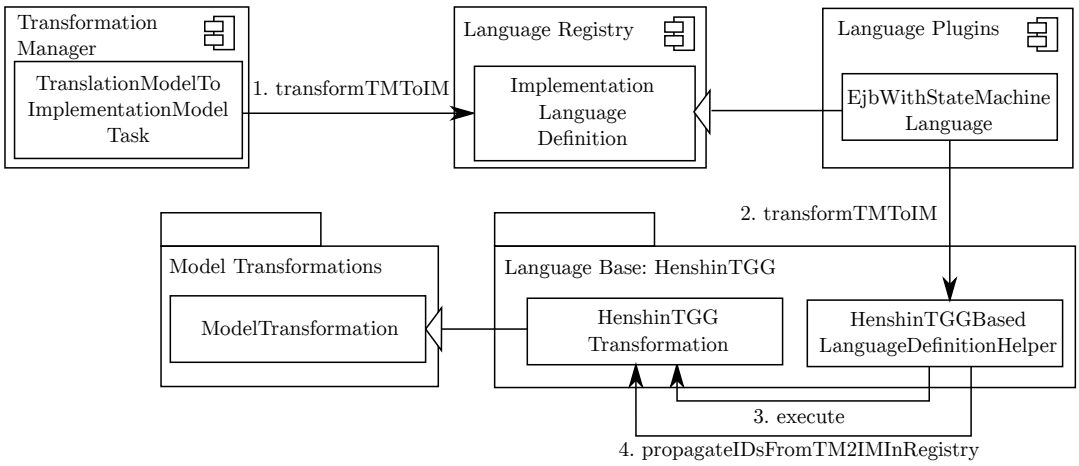


Figure 9.23: An overview of the messages between the different part of the Codeling implementation during step 6 and their order

```
public class EjbWithStateMachineLanguage extends
    JavaBasedImplementationLanguageDefinition {

  final URI henshinTGGFileURI = URI.createPlatformPluginURI(
      "/" + FrameworkUtil.getBundle(getClass()).getSymbolicName() + "/AIL2IAL.henshin",
          true);

  [...]

  @Override
  public TransformationResult transformTMToIM(List<EObject> ilRoots, IDRegistry
      idRegistry) throws CodelingException {
    return new HenshinTGGBasedLanguageDefinitionHelper().transformILToCM(this,
        henshinTGGFileURI, ilRoots, idRegistry, monitor);
  }

  [...]

}
```

Listing 9.22: An excerpt of the implementation language definition type for the running example, that triggers the translation from the translation model to the implementation model with HenshinTGG

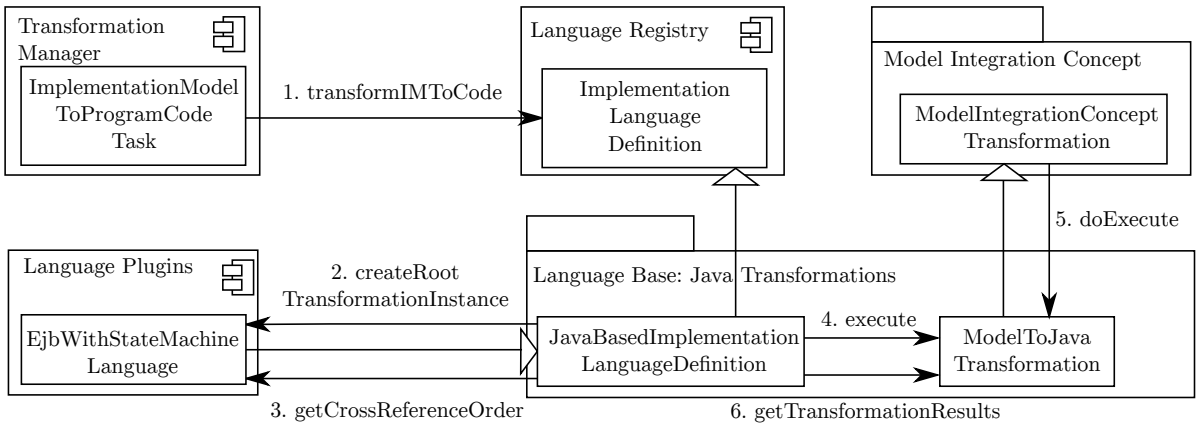


Figure 9.24: An overview of the messages between the different part of the Codeling implementation during step 6.2 and their order

Step 6.2 – Implementation Model to Program Code

In step 6.2 the changes in the implementation model are propagated to the code. The task `ImplementationModelToProgramCodeTask` takes the prior ID registry from the result of step 1.1 (Program Code to Implementation Model), the ID registry and implementation model of the result of step 6.1, and the selected projects in the IDE as input. It uses the method `transformIMToCode` of the type `ImplementationLanguageDefinition` to execute the transformation (see Figure 9.24).

This method is implemented in the type `JavaBasedImplementationLanguageDefinition`. It uses the framework for bidirectional Java-to-Model transformations in the Java Transformations library, which has already been used in the steps 1.1 (Program Code to Implementation Model) and 1.3 (Program Code to Translation Model). First, it requests a root transformation instance from the implementation language definition (see Listing 9.13) and configures it with the aforementioned ID registry instances, models, and projects. The root transformation and all its children are executed using the type `ModelToJavaTransformation`, a subtype of `ModelIntegrationConceptTransformation` from the Model Integration Concept library (see Step 1.1 – Program Code to Implementation Model).

The Model-to-Java transformation is the opposite of the Java-to-Model transformation, in step 1.1. It takes a list of project paths, a root transformation instance, and an ordering of classes in the targeted language meta model to determine the order in which non-containment references should be translated. During execution, first the program code is updated according to the model, by executing the root transformation with the framework for bidirectional Java-to-Model transformations. The framework uses the same transformation types that were used in step 1.1, and executes them concurrently in Java's `ForkJoinPool`. Figure 9.25 shows an excerpt of the type hierarchy of these transformations in the context of the running example, with a focus on the Model-to-Java direction. The types and the shown attributes and methods will be described in the remainder of this section.

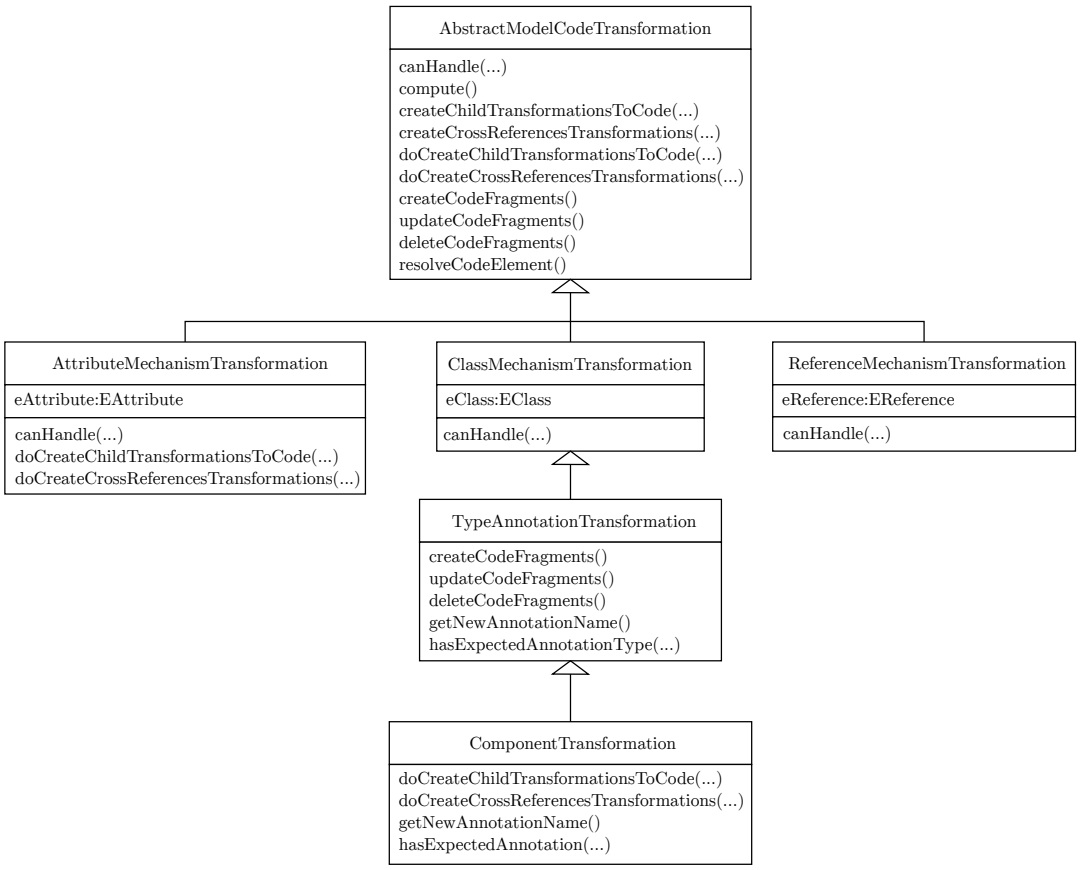


Figure 9.25: Excerpt of the type hierarchy of transformations in the bidirectional model-code transformation framework, with attributes and methods for the steps 6.2 and 6.3

The abstract root transformation type `AbstractModelCodeTransformation` implements the process for each single transformation type. Listing 9.23 shows this implementation. Logging and error handling has been removed in this listing for readability reasons.

```
private void computeTransformationToCode() {                                    1
  if (!onlyCrossReferences) {                                                   2
    if (modelElement == null) { // The model element has been deleted           3
      resolveCodeElement();                                                     4
      deleteCodeFragments();                                                    5
    } else if (priorModelElement == null) { // The model element is new         6
      createCodeFragments();                                                    7
    } else { // The model element might have changed                            8
      resolveCodeElement();                                                     9
      updateCodeFragments();                                                    10
    }                                                                           11
                                                                                12
    // Execute child transformations.                                           13
    // ReferenceMechanismTransformation already do this themselves, because they use  14
    // the results of the child transformations. Therefore we only execute      15
    // childTransformations that do not already exist.                          16
    if (childTransformations == null || childTransformations.isEmpty()) {      17
      childTransformations = createChildTransformationsToCode();               18
      childTransformations.forEach(c -> c.setOnlyCrossReferences(onlyCrossReferences));19
      invokeAll(childTransformations);                                          20
    }                                                                           21
                                                                                22
    // Registers all class, reference, and attribute transformations for an Eclass.  23
    EClass eClass = modelElement == null ? priorModelElement.eClass() : modelElement.  24
        eClass();
    findTranslatedElements.addTransformation(eClass, this);                     25
  } else {                                                                      26
    final List<AbstractModelCodeTransformation<?, ?>> crossReferenceTransformations =  27
        createCrossReferencesTransformations();
    crossReferenceTransformations.forEach(t -> t.setToCode());                  28
    invokeAll(crossReferenceTransformations);                                   29
  }                                                                             30
}                                                                               31
                                                                                32
public JAVAELEMENTCLASS resolveCodeElement() throws CodelingException {         33
  if (priorIDRegistry != null && priorModelElement != null)                     34
    codeElement = (JAVAELEMENTCLASS) priorIDRegistry.getCodeElementFromComponentModel(35
        priorModelElement);
  else                                                                          36
    codeElement = (JAVAELEMENTCLASS) idRegistry.getCodeElementFromComponentModel(  37
        modelElement);
  return codeElement;                                                           38
}                                                                               39
```

Listing 9.23: Excerpt of the bidirectional Model-to-Java transformation for the *Component* class in the meta model of the running example

During the first execution the flag `onlyCrossReferences` is false. First it is evaluated whether the model element, that is subject to translation, has been deleted or created. If none of that is true, it might be changed. The transformation instance can have a model element, i.e. the element resulting from step 5, and a prior model element, i.e. the element resulting from step 1.1. When the model element has been deleted, a transformation instance exists without a model element, but with a prior model element. In that case the code element is resolved from the prior ID registry, and the method `deleteCodeFragments` of the specific transformation type is called. When the model element has been created, a transformation

instance exists without a prior model element, but with a model element. In that case the method `createCodeFragments` of the specific transformation is called. When both, a model element and a prior model element exist, first the code element is resolved from the ID registry, and the method `updateCodeFragments` of the specific transformation is called. After the translation, the child transformation instances are requested from the specific transformation, and added to the `ForkJoinPool`. Afterwards the transformation instance is registered in a transformation registry, so that it can be found by other transformations. This is necessary when the translation of a model element depends on the type of translation of another element, and serves as an in-memory mapping between prior model elements, model elements, and code elements during the transformation.

When all transformations for model objects, their attributes, and containment references are executed, the `ModelToJavaTransformation` executes all transformations again, with the flag `onlyCrossReferences`. During this translation, for all model objects, child translations for non-containment references are requested and added to the `ForkJoinPool`. These transformations are executed at last, so it is ensured that the target element of each reference has already been created.

Listing 9.24 shows an excerpt of the `ComponentTransformation` of the running example, which implements the methods necessary for the model-to-code transformation. The code-to-model part of the type has already been shown in step 1.1 in Listing 9.14. It extends the `TypeAnnotationTransformation` for the *Component* class in the example meta model. The method `doCreateChildTransformationsToCode` add instances of child transformations to a result list, for the attributes (none in the example) and containment references owned by the *Component* class. The method `doCreateCrossReferencesTransformations` adds an instance of the transformation for the non-containment reference called *references*.

```
public class ComponentTransformation extends TypeAnnotationTransformation<Component> {
  @Override
  public void doCreateCrossReferencesTransformations(
      List<AbstractModelCodeTransformation<? extends EObject, ? extends IJavaElement>>
          result) {
    result.add(new ReferencesTransformation(this));
  }

  @Override
  protected void doCreateChildTransformationsToCode(
      List<AbstractModelCodeTransformation<? extends EObject, ? extends IJavaElement>>
          result) {
    result.add(new StatemachineTransformation(this));
    result.add(new OperationsTransformation(this));
  }

  @Override
  protected String getNewAnnotationName() {
    return "javax.ejb.Stateful";
  }

  [...]
}
```

Listing 9.24: The implementation for propagating model changes to Java code in the type `AbstractModelCodeTransformation`

The method `getNewAnnotationName` configures the supertype `TypeAnnotationTransformation`. Because the notation for the *Component* does not exactly match the Type Annotation mechanism, another annotation name has to be stated here. Listing 9.25 shows the excerpt of this abstract transformation type. In the method `createCodeFragments` the transformation first ensures that a default package exists, in which the resulting type declaration can be placed. When a package declaration exists, that represents the model element's parent (references it via a containment reference), this package is chosen. It then collects all necessary information from the model element. It creates the program code that represents the model element using the Type Annotation mechanism via an Xtend template expression with the identifier `content`, and writes it to a Java file. In Xtend, template expressions can be defined using three apostrophes as delimiters. Within these template expressions, control flow commands and variables can be used between guillemets («expression») to build the desired string. In first declares the package for the type and imports the annotation used in the notation. It then declares a type with the *name* attribute value of the model element as type name. The annotation used by the Type Annotation model notation is created using the method `getNewAnnotationName`. By default this is an annotation with the name of the class, declared in a specific package. In the running example, this method is overridden to use the annotation `javax.ejb.Stateful`, as required by the targeted implementation language. The method `updateCodeFragments` compares two model elements, one from before and one from after the model changes. It compares the model elements and changes the code accordingly, using refactoring mechanisms where applicable. In the given example, a renaming refactoring is executed on the Java type, which represents the model element, when the name changed. When the code element has to be deleted, the method `deleteCodeFragments` deletes the complete compilation unit. After the model changes have been propagated to the program code, the ID registry is updated accordingly.

```
abstract class TypeAnnotationTransformation<ELEMENTECLASS extends EObject> extends
    ClassMechanismTransformation<ELEMENTECLASS, IType> {

  override createCodeFragments() throws CodelingException {
    var IPackageFragment targetPackage = if (parentCodeElement !== null &&
        parentCodeElement instanceof IType)
        (parentCodeElement as IType).packageFragment
      else
        createDefaultPackage();

    val String name = modelElement.nameAttributeValue;
    val String typeName = name.toFirstUpper;
    val String annotationName = getNewAnnotationName().substring(getNewAnnotationName()
        .lastIndexOf(".") + 1);
    val String annotationPackage = getNewAnnotationName().substring(0,
        getNewAnnotationName().lastIndexOf("."));

    val String content = '''
␣␣␣␣␣␣package␣«targetPackage.elementName»;

␣␣␣␣␣␣import␣«annotationPackage».«annotationName»;

␣␣␣␣␣␣@«annotationName»
␣␣␣␣␣␣public␣class␣«typeName»␣{}''';
    val ICompilationUnit cu = targetPackage.createCompilationUnit(typeName + ".java",
        content, true, null);
    codeElement = cu.getType(typeName);
  }

  override updateCodeFragments() throws CodelingException {
    // Update of name attribute
    val String oldName = priorModelElement.nameAttributeValue;
    val String newName = modelElement.nameAttributeValue;

    if (!newName.equals(oldName)) {
      ASTUtils.renameType(codeElement, newName); // Execute renaming refactoring
      codeElement = codeElement.getPackageFragment().getCompilationUnit(newName + ".
          java").getType(newName);
    }
  }

  override deleteCodeFragments() {
    codeElement.compilationUnit.delete(true, null);
  }

  protected def String getNewAnnotationName() {
    return '''org.codeling.lang.«getLanguageName».mm.«eClass.name.toFirstUpper»''';
  }

  [...]
}
```

Listing 9.25: An excerpt of the abstract bidirectional Java-to-Model transformation for the Type Annotation mechanism, implemented with the Xtend programming language

Step 6.3 – Translation Model to Program Code

The task for step 6.3 of the process uses the method `transformTMToCode` of the type `ImplementationLanguageDefinition` to propagate changes in the translation model to the program code, when the information cannot be represented with the architecture implementation language. Figure 9.26 gives an overview of the messages during the execution of this task. The IAL transformations in the framework for bidirectional Java-to-Model transformations is used for these translations, analogously to step 1.3.

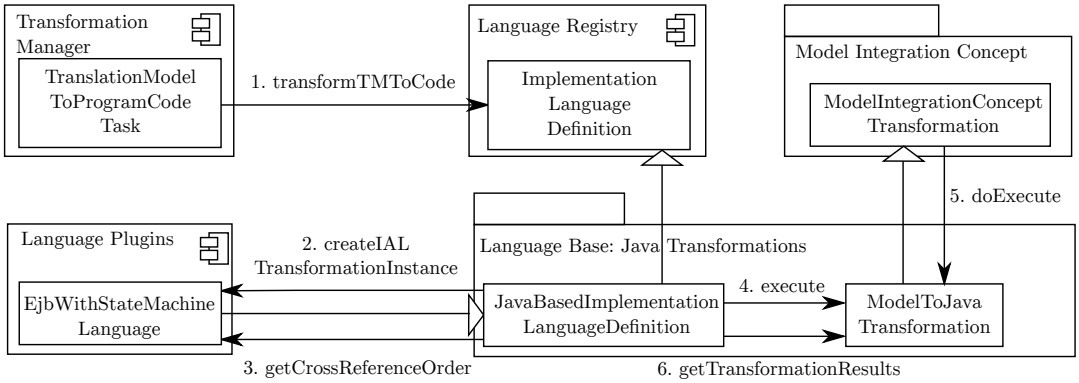


Figure 9.26: An overview of the messages between the different part of the Codeling implementation during step 6.3 and their order

The type `JavaBasedImplementationLanguageDefinition` implements the method `transformTMToCode` (see Listing 9.26). It takes the following input: the implementation models of step 6.2 and step 1.1, the translation models of step 5 and step 3, the ID registry of step 6.2, the id registry of step 1.3, and the list of projects as input. For each entry in the ID registry the element of the original models, the changed models, and the program code are mapped via the id registries. Then for each implementation model element, IAL transformation instances are created, configured, and executed using the `ModelToJavaTransformation` type as executor, following the same process as the translation in step 6.2.

During this step, the IAL transformation types of step 1.3 are reused. They are declared in the implementation language definition as described in step 1.3 (see Listing 9.18).

9.2.4 Further Use Cases

Section 9.2.3 described how an architecture specification model is extracted in Codeling, and how changes in the model are propagated to the program code. Codeling can also serve in other use cases.

During the life time of a long-living system, it often becomes necessary to migrate existing software to a new architecture implementation language, e.g. because the platform for the system is not maintained anymore. To support **implementation migrations**, Codeling allows to choose an implementation language instead of a specification language as target for a translation. In this case, the process is changed. Step 1.1 to 1.3 (Program Code

```
public abstract class JavaBasedImplementationLanguageDefinition extends
    ImplementationLanguageDefinition {
  [...]

  @Override
  public void transformTMToCode(List<EObject> newIMRoots, List<EObject> oldIMRoots,
      List<EObject> newTMRoots,
      List<EObject> oldTMRoots, IDRegistry idRegistry_newModel, IDRegistry
          idRegistry_oldModel,
      List<IJavaProject> projects) throws CodelingException {

    idRegistry.getRegistryEntries().keySet().stream().forEach(id -> {
      final EObject imElement = idRegistry_newModel.resolveImplementationModelElement(
          id, newIMRoots); [...]
      final List<? extends IALTransformation<?, ?>> ialTransformations =
          createIALTransformationInstance(imElement); [...]

      final EObject foundationalIALElement = idRegistry_newModel.
          resolveTranslationModelElement(id, newTMRoots);
      final EObject priorFoundationalIALElement = idRegistry_oldModel.
          resolveTranslationModelElement(id, oldTMRoots);
      final IJavaElement codeElement = idRegistry_newModel.getCodeElement(id);

      if (ialTransformations != null) {
        ialTransformations.stream().forEach(t -> {
          t.setIDRegistry(idRegistry_newModel);
            t.setPriorIDRegistry(idRegistry_oldModel);
            t.getIALHolder().setFoundationalIALElement(foundationalIALElement);
            t.getIALHolder().setPriorFoundationalIALElement(priorFoundationalIALElement
                );
            t.getIALHolder().setIALCodeElement(codeElement);
            t.getIALHolder().setIALRoots(newTMRoots);
            StereotypeApplication modelElement = t.resolveTranslatedIALElement(
                foundationalIALElement);
            StereotypeApplication priorModelElement = t.resolveTranslatedIALElement(
                priorFoundationalIALElement);
            if (modelElement == null && priorModelElement == null)
              return; // No new, existing, or deleted element

            t.setModelElement(modelElement);
            t.setPriorModelElement(priorModelElement);

            final ModelToJavaTransformation executor = new ModelToJavaTransformation(
                projects, (AbstractModelCodeTransformation<?, ?>) t, new LinkedList<>()
                );
            executor.execute(monitor);
        });
      }
    });

    [...]
  }

  [...]
}
```

Listing 9.26: An excerpt of the generic implementation for propagating changes from the translation model to the program code

to Translation Model via Implementation Model) are executed as shown above. In step 2 (Inter-Profile Transformations), the target language is not the specification language, but an implementation language. As no specification language is involved, the steps 3 (Transformation Model to Specification Model), 4 (Specification Model to Translation Model), and 5 (Inter-Profile Transformations) are not executed. After the adapted step 2, a translation model exists, that is prepared for the target implementation language. Then a new project is created in the IDE, and a new ID registry is created, effectively abandoning all references to an implementation model or program code. When the steps 6.1 to 6.3 (Translation Model to Program Code via Implementation Model) are executed, new program code for the target implementation language is created. In the current state of the implementation, the contents of the entry points—e.g. method bodies—are not transferred to the new implementation. Only program code is created, which is part of the extracted architecture model.

Codeling can be the basis for **architecture compliance checking**. Usually it is desired to implement the architectural concepts, e.g. components, with the same programming style. E.g. a component should always be implemented in the same way. Codeling can support such a compliance checking. By design, Codeling requires program code to comply with specific programming styles for expressing architectural concepts. When a component is implemented in a way, that is not part of the defined translations, it will not be translated to the specification model.

Imagine a development scenario where Codeling is automatically executed upon program code committed to a source code management system by a development team. For each committed version, an architecture specification model is created and visually displayed, so that the whole development team can see the architecture as it is implemented. When a component is not developed following the rules for creating components as specified by the transformations in Codeling, the component will not exist in the resulting specification model. It will not be announced and presented to the development team.

The use of Codeling in this example can also be beneficial for detecting unwanted connections between components. Imagine a layered system, in which architectural constraints define that operation calls are only allowed from components in upper layers to components in the same or lower layer. When operation calls are implemented, that violate this rule, the violation can be detected in the specification model. The integration of architectural constraints in the Context of Codeling has been the subject to the master's thesis of Enno Lohmann [Loh17].

9.2.5 Extensibility

Codeling is meant to be extended. Therefor it provides a defined architectural structure (see Section 9.2.2) and extension points. Codeling provides an extension point for language plugins. Language plugins implement transformations for an architecture implementation or specification language in Codeling. They can use the language base libraries provided in this context for making the development of transformations easier. They include abstractions for executing Henshin or HenshinTGG transformations, and a framework for bidirectional model-code transformations, that follow the Model Integration Concept. For adding new language plugins to Codeling, Eclipse plugins must be included in an Eclipse instance, that specify an extension of the Eclipse extension point `org.codeling.languageRegistry`. The extension must declare a human readable name for the language, a symbolic name, a version, and a Java type, that implements a language definition. The language will be registered at the language registry.

The current implementation of Codeling includes the framework for bidirectional model-code transformations based on Java and Ecore. Codeling provides the Model Integration Concept library with an abstract type `ModelIntegrationConceptTransformation`, that is used by the Transformation Manager component to execute a model-to-code or code-to-model transformation. The framework implements this abstract type. For extending Codeling with further types of model-code transformations, further implementations of this abstract type could be provided in an Eclipse plugin. Translation frameworks for architecture implementation languages, which are not based on Java byte code, need to implement at least a compatible type declaration, e.g. for executing native calls to operations in C++ or Python.

Analogously to the Model Integration Concept library, Codeling provides the Model Transformation library with an abstract type `ModelTransformation` for executing model transformations. These transformations can be used for inter-profile transformations and for translations between the IAL and architecture implementation or specification languages. In the current implementation the Henshin and HenshinTGG libraries extend the type `ModelTransformation`. Further extensions could be implemented to make use of other model transformation languages, such as QVT [Obj08] or ATL [JABK08].

Finally, the user interface of Codeling is tightly coupled to the Eclipse platform, by adding menu items to the *package explorer* view. The user interface triggers translation tasks, that are provided by the Transformation Manager component. The tasks are independent from the user interface, therefore other user interfaces can be implemented. New user interfaces should also trigger the tasks. This could be used e.g. to extract an architecture specification model automatically in the context of a continuous integration server [DMG07]. This would make the resulting model available for continuous analysis and communication.

9.3 Code Generation Tool for Integration Mechanisms

Codeling provides libraries and components for defining and executing translations as defined by the Explicitly Integrated Architecture approach. This includes the definition of bidirectional model-code transformations using the corresponding framework, and meta model notation libraries, that e.g. include the annotations used by the transformations. Developing such transformations and meta model notation libraries can be cumbersome and error-prone. The integration mechanisms can be used as templates for generating transformations for model notations and program code libraries for meta model notations. Section 5.7 describes how languages can be integrated with the Model Integration Concept. With a mapping of meta model elements to integration mechanisms, program code for three aspects of the Explicitly Integrated Architecture approach can be generated. The code generation tool for integration mechanisms is an Eclipse plugin, which supports this integration by an automated code generation based on meta models and integration mechanisms as follows: **(1)** It is possible to automatically generate program code structures for meta model notations (see Section 9.3.3). **(2)** For the design time of architectures, transformations can be generated, that translate between a model view and a code view of model elements (see Section 9.3.4). **(3)** For the run time of architectures, execution runtime stubs for integrated model elements can be generated (see Section 9.3.5). After describing the code generation, we describe how to implement new integration mechanisms for Codeling and the code generation tool (see Section 9.3.6), and how to integrate the generated code with Codeling (see Section 9.3.7).

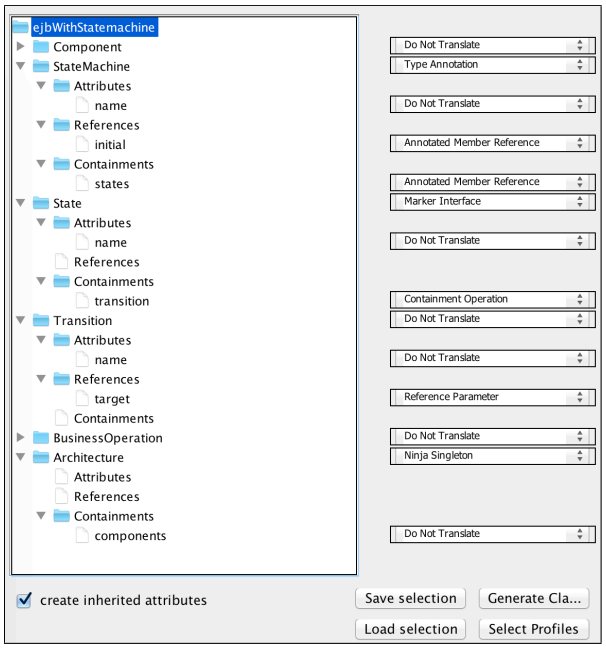


Figure 9.27: User Interface of the Code Generator for Integration Mechanisms

9.3.1 Use Case – Generating Code for the Running Example

The following actions have to be taken to generate meta model notation libraries, transformations for the model notations, and corresponding execution runtime stubs for the running example (see Section 9.1):

1. A meta model has to be created using Ecore. The meta model of the running example is shown in Figure 9.2. The program code for programmatically managing models of this meta model has to be created using the Ecore tools, as it is typically done for Ecore models (see [SBPM09, Chapter 12])

2. A mapping of meta model elements to integration mechanisms has to be created. The code generation tool provides means to create this mapping. Figure 9.27 shows an excerpt of the running example's mapping in the code generation tool.

3. The code for meta model elements (see Section 9.3.3), transformations (see Section 9.3.4), and an execution runtime (see Section 9.3.5) has to be generated using the code generation tool.

4. The execution runtime types need to be extended with the specific execution semantics. Section 9.3.5 describes how the generated execution runtime is extended with execution semantics for the state machine class in the running example.

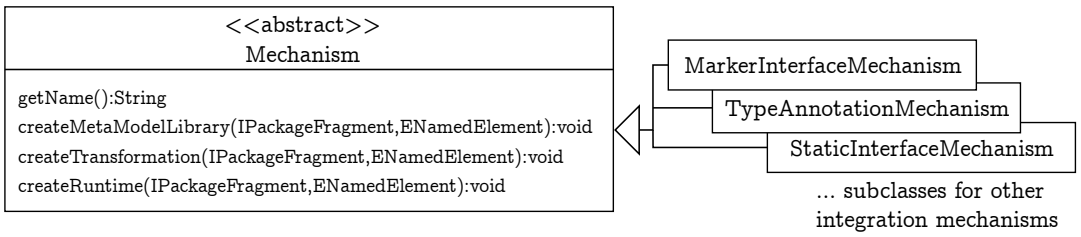


Figure 9.28: UML class diagram of the generation part in the code generation tool prototype

9.3.2 User Interface and Design

In the code generation tool, a mapping of meta model elements to integration mechanisms can be created, and the code generation can be triggered. Figure 9.27 shows the prototype's user interface. The left hand side shows the package of the Ecore meta model in Figure 9.2 as the root of a tree. All classes are listed in the level below the root. Each class in the tree contains its attributes, non-containment references, and containment references. For each class, attribute, and reference, the right hand side shows the selected integration mechanisms out of a list of integration mechanisms, that are applicable for the associated type of meta model element. A list for a class contains all integration mechanisms for representing classes.

The code generation tool uses the mechanism type hierarchy, which has been shown for Codeling in Figure 9.14. Figure 9.28 shows an excerpt of the type hierarchy, with a focus on the use in the code generation tool. The abstract type *Mechanism* declares abstract methods for the subtypes to implement. The actual code generation of each integration mechanism is implemented in a subtype. In each subtype, the method `getName` returns the name of the mechanism. The method `createMetaModelLibrary` creates the meta model notation program code (see Section 9.3.3). The method `createTransformation` creates the code that realizes the bidirectional transformation between the code and the model representation of the model notations (see Section 9.3.4). The method `createRuntime` creates runtime stubs for the meta model elements (see Section 9.3.5). The code generating methods take two parameters: The parameter of the type `IPackageFragment` is a reference to a package into which the program code will be generated. The parameter of the type `ENamedElement` is the meta model element for which code is generated. The execution of these methods result in generated program code.

9.3.3 Meta Model Notations

Most integration mechanisms in this thesis declare meta model notations. The program code structures within the meta model notations include interface and annotation declarations. These declarations are reusable between the code representation of different model elements. E.g. in the running example, multiple Java types represent components by reusing the same annotation `Stateful`. The prototype tool generates these code structures for meta model notations, so that they are available as maven[3] project. The generated code can be used as a Java library.

Listing 9.27 shows, as an example, the implementation of the method `createMetaModelLibrary` for the Type Annotation mechanism. The code generation is

[3]Apache's build automation framework for Java – `https://maven.apache.org`

implemented with the Xtend programming language using Xtend's template expressions. In the listing, the method creates an annotation named after the meta model element. First the annotation name is derived from the name of the translated class in the meta model. Next, the content of the resulting program code file is generated. An annotation with the given name is declared within the package given as parameter. The meta annotations attached to the annotation declaration are signals for the Java compiler to make the declared annotation available at runtime. At last a program code file is created via Eclipse's JDT[4] in the workspace of the running Eclipse IDE.

```
override createMetaModelLibrary(                                                    1
        IPackageFragment packageFragment, ENamedElement element) {                  2
  var String annotationName = element.name.toFirstUpper;                            3
                                                                                    4
  val content = '''                                                                 5
␣␣␣␣package␣«packageFragment.FQN»;                                                  6
                                                                                    7
␣␣␣␣@java.lang.annotation.Retention(java.lang.annotation.RetentionPolicy.RUNTIME)   8
␣␣␣␣@java.lang.annotation.Target(java.lang.annotation.ElementType.TYPE)           9
␣␣␣␣public␣@interface␣«annotationName»␣{                                           10
                                                                                    11
␣␣␣␣}''';                                                                          12
                                                                                    13
  return packageFragment.createCompilationUnit(                                    14
          '''«annotationName».java''', content, true, monitor).getType(annotationName);15
}                                                                                   16
```

Listing 9.27: Generation of meta model notation code for the Type Annotation mechanism using the Xtend programming language

9.3.4 Transformations

The code generation tool generates bidirectional model-code transformations. The generated transformations make use of a generic transformation framework for Java-based architecture implementation languages, that has been developed for Codeling (see step 1.1 in Section 9.2.3). The framework includes an hierarchy of abstract transformation types. The type `AbstractModelCodeTransformation` is the basis for all transformations. It includes fields for model and code elements, provides abstract methods for creating, updating, and deleting code elements based on model elements, and a method for creating model elements based on code. It also provides the choice which of these methods are called during translation. Abstract transformation types exist for classes, attributes, containment, and non-containment references in a meta model.

The tool generates specific transformations for each pair of meta model element and associated integration mechanism. The generated transformations extend the abstract transformations of the corresponding integration mechanism. The specific transformations configure their abstract supertype with their specific meta model element to be translated, and include code for initializing child transformations. The children of class transformations are attribute and reference transformations. Children of reference transformations are the class transformations of their targets.

[4]Eclipse Java Development Toolkit – `https://www.eclipse.org/jdt/`

Listing 9.28 shows an excerpt of the generated transformation for the state machine class in the running example. The transformation Java type extends the abstract transformation for the Type Annotation mechanism from Listing 9.15. The supertype is configured with the `StateMachine` annotation as type parameter. This is the meta model representation of the class *StateMachine* in the meta model, generated with the same tool (see Section 9.3.3). In the methods `doCreateChildTransformationsToCode` and `doCreateChildTransformationsToModel`, the child transformations are created and configured for containment references and attributes. In the method `doCreateCrossReferencesTransformations`, the child transformations for non-containment references are created.

```
public class StateMachineTransformation extends TypeAnnotationTransformation<
    StateMachine> {
  public StateMachineTransformation(
      AbstractModelCodeTransformation<? extends EObject, ? extends IJavaElement>
          parentTransformation) {
    super(parentTransformation, ejbWithSMPackage.eINSTANCE.getStateMachine());
  }

  @Override
  public void doCreateCrossReferencesTransformations(
      List<AbstractModelCodeTransformation<? extends EObject, ? extends IJavaElement>>
          result) {
    result.add(new InitialTransformation(this));
  }

  @Override
  protected void doCreateChildTransformationsToCode(
      List<AbstractModelCodeTransformation<? extends EObject, ? extends IJavaElement>>
          result) {
    result.add(new StatesTransformation(this));
  }

  @Override
  protected void doCreateChildTransformationsToModel(
      List<AbstractModelCodeTransformation<? extends EObject, ? extends IJavaElement>>
          result) {
    result.add(new StatesTransformation(this));
  }
}
```

Listing 9.28: Excerpt of the generation of transformations for the Type Annotation mechanism

Listing 9.29 shows an excerpt of the code generator for specific transformations of the Type Annotation mechanism. First, the necessary information is collected. Then the transformation code is generated, and a Java program code file is created. The code generators share code for constructing transformations for containment and non-containment references.

9.3.5 Execution Runtimes

The architectural elements described with the architecture implementation and specification languages are development time representations of architectural concepts. All of these concepts have run time representations. E.g. component types are instantiated at run time, meaning that an instance exists, and its operation-based provisions can be invoked. A generic run time

```
override createTransformation(IPackageFragment packageFragment, ENamedElement element)
    {
  val String typeName = element.name.toFirstUpper;
  val EClass eClass = element as EClass;
  val String eClassName = eClass.name;
  val String eClassInstantiation = getEClassInstantiation(eClass);

  val content = '''
␣␣␣␣package␣«packageFragment.FQN»;

␣␣␣␣//...␣imports

␣␣␣␣public␣class␣«typeName»Transformation␣extends␣TypeAnnotationTransformation<«
    eClassName»>␣{

␣␣␣␣␣␣public␣«typeName»Transformation(AbstractModelCodeTransformation<?␣extends␣EObject
    ,␣?␣extends␣IJavaElement>␣parentTransformation)␣{
␣␣␣␣␣␣␣␣super(parentTransformation,␣«eClassInstantiation»);
␣␣␣␣␣␣}

␣␣␣␣␣␣«createCrossReferencesTrasformations»
␣␣␣␣␣␣«createChildTransformationsToCode»
␣␣␣␣␣␣«createChildTransformationsToModel»
␣␣␣␣}''';

  packageFragment.createCompilationUnit('''«typeName»Transformation.java''', content,
      true, monitor);
}
```

Listing 9.29: Excerpt of the generation of transformations for the Type Annotation mechanism using the Xtend programming language

framework has been implemented as a prototype, for meta models that have notations based on integration mechanisms.

Analogously to the transformations, generic execution runtimes for notations based on integration mechanisms have been implemented. These runtimes are based on Java's reflection mechanism to analyse the code, instantiate types, inject objects into fields, and invoke operations. Listing 9.30 shows an excerpt of the generic execution runtime for the Type Annotation mechanism as an example. The runtime's constructor takes the implementing type and the annotation type as parameter. The method `initialize` first verifies that the type is actually an instance of the Type Annotation mechanism, before it instantiates the type and stores the instance. The instance can be obtained by a `getInstance` operation of the supertype. A registry named `Runtimes` stores a map of all objects to their runtimes.

The code generation tool can be used to generate runtimes for specific meta model elements, based on the mapping from meta model elements to integration mechanisms. The generated runtimes each extend one of these generic integration mechanism runtimes. Listing 9.31 shows an excerpt of the code generation for runtimes of the Type Annotation mechanism. The method `createRuntime` derives the type name of the runtime from the meta model element's name. Then it creates the content of the runtime type using a template expression. The generated runtime extends the generic runtime for the Type Annotation mechanism. It has fields for its attribute and reference runtimes, a constructor, and methods for initializing its containment reference runtimes and its non-containment reference runtimes. At last a program code file is

```
public class TypeAnnotationRuntime<T> extends TypeMechanismRuntime {
  private final Class<T> implementingClass;
  private final Class<? extends Annotation> typeAnnotation;

  public TypeAnnotationRuntime(Class<T> implementingClass,
                Class<? extends Annotation> typeAnnotation) { //... set fields }

  @Override
  public void initialize() throws IntegratedModelException {
    verifyTypeAnnotationMechanism();

    try {
      instance = implementingClass.getConstructor(new Class[0]).newInstance(
        new Object[0]);
      Runtimes.getInstance().put(instance, this);
    } catch (final Exception e) {
      throw new IntegratedModelException([...], e);
    }
  }

  private void verifyTypeAnnotationMechanism() throws IntegratedModelException {
    if (!implementingClass.isAnnotationPresent(typeAnnotation))
      throw new IntegratedModelException([...]);
  }
}
```

Listing 9.30: Excerpt of the generic Type Annotation runtime

created with the contents.

The actual execution semantics of the elements must be implemented within the generated runtime types by extending them with corresponding operations. Listing 9.32 shows an excerpt of generated runtime code for the class *StateMachine* in the running example, which extends the Type Annotation runtime shown in Listing 9.30. The runtime code contains a constructor that takes the state machine type reference as input—identified by the corresponding annotation. It configures the generic runtime with that type and with the state machine annotation. The state machine has no attribute beside its *name* attribute, so there is no attribute runtime to be instantiated. In the method `initializeContainments`, the runtime for its containment reference *states* is created and initialized. In the method `initializeCrossReferences`, the runtime for its non-containment reference *initial* is created and initialized. These parts have been generated by the prototype tool. The rest of the program code defines the execution semantics, which have been manually added. After initializing the runtime for the reference *initial*, the initial state runtime is stored as the runtime for the current state. The method `executeTransition` takes a transition name as input. It first validates whether the desired transition exists and is executable. Then it takes the runtime for the containment reference *transitions* of the current state, a Containment Operation runtime. It invokes the containment operation that represents the transition, and sets the current state the instance of the target state.

```
override createRuntime(IPackageFragment packageFragment, ENamedElement element) {
  val String typeName = element.name.toFirstUpper;

  val content = '''
␣␣␣␣package␣«packageFragment.FQN»;

␣␣␣␣//...␣imports

␣␣␣␣public␣class␣«typeName»Runtime<T>␣extends␣TypeAnnotationRuntime<T>␣{

␣␣␣␣␣␣«super.attributeRuntimeFields»
␣␣␣␣␣␣«super.referenceRuntimeFields»

␣␣␣␣␣␣public␣«typeName»Runtime(Class<T>␣implementingClass)␣throws␣
    IntegratedModelException␣{
␣␣␣␣␣␣␣␣super(implementingClass,␣«typeName».class);
␣␣␣␣␣␣}

␣␣␣␣␣␣@Override
␣␣␣␣␣␣public␣void␣initializeContainments()␣throws␣IntegratedModelException␣{
␣␣␣␣␣␣␣␣«super.attributeRuntimeInitialization»
␣␣␣␣␣␣␣␣«super.containmentReferenceRuntimeInitialization»
␣␣␣␣␣␣}

␣␣␣␣␣␣@Override
␣␣␣␣␣␣public␣void␣initializeCrossReferences()␣throws␣IntegratedModelException␣{
␣␣␣␣␣␣␣␣«super.crossReferenceRuntimeInitialization»
␣␣␣␣␣␣}

␣␣␣␣␣␣«super.getReferenceRuntimeFieldGetters»

␣␣␣␣}''';

  packageFragment.createCompilationUnit('''«typeName»Runtime.java''', content, true,
      monitor);
}
```

Listing 9.31: Excerpt of the generation of runtime code for the Type Annotation mechanism using the Xtend programming language

```
public class StateMachineRuntime<T> extends TypeAnnotationRuntime<T> {

  // Reference Runtimes
  InitialRuntime initialRuntime;
  StatesRuntime statesRuntime;
  //... getters for initialRuntime and statesRuntime

  public StateMachineRuntime(Class<T> implementingClass) throws
      IntegratedModelException {
    super(implementingClass, StateMachine.class);
  }

  @Override
  public void initializeContainments() throws IntegratedModelException {
    // Reference Runtimes
    statesRuntime = new StatesRuntime(this);
    statesRuntime.initialize();
  }

  @Override
  public void initializeCrossReferences() throws IntegratedModelException {
    // Reference Runtimes
    initialRuntime = new InitialRuntime(this);
    initialRuntime.initialize();

    // Set current state (not generated)
    currentStateRuntime = (StateRuntime<?>) initialRuntime.getTargetRuntime(
        initialRuntime.getTargets()[0]);
  }

  // Execution semantics - not generated
  StateRuntime<?> currentStateRuntime;

  public StateRuntime<?> getCurrentStateRuntime() {
    return currentStateRuntime;
  }

  public void executeTransition(String transitionName) throws IntegratedModelException
      {
    if (!isExecutable(transitionName))
      throw new IllegalStateException([...]);
    TransitionRuntime transitionRuntime = currentStateRuntime.getTransitionRuntime();
    transitionRuntime.invoke(transitionName);
    StateRuntime<?> targetStateRuntime = (StateRuntime<?>) transitionRuntime.
        getTargetRuntime().getTargetRuntime(transitionName)[0];
    currentStateRuntime = (State) targetStateRuntime.getInstance();
  }

  public boolean isExecutable(String transitionName) throws IntegratedModelException {
    return getPossibleTransitions().contains(transitionName);
  }

  public List<String> getPossibleTransitions() throws IntegratedModelException {
    TransitionRuntime transitionRuntime = currentStateRuntime.getTransitionRuntime();
    List<Method> containmentOperations = transitionRuntime.getContainmentOperations();
    return containmentOperations.stream().map(Method::getName).collect(Collectors.
        toList());
  }

}
```

Listing 9.32: Excerpt of the runtime for a State Machine class

The running example defines that the invocation of the component operation of the *CashDesk* component triggers a transition in the state machine. Listing 9.33 shows how an EJB implementation of the *CashDesk* component makes use of the state machine's execution runtime to implement this behaviour. A complete implementation of runtimes for the running example and their use can be found on the data medium attached to this thesis (see Appendix B).

```
@Stateful
public class CashDesk {
  final LinkedList<String> items = new LinkedList<>();
  StateMachineRuntime smr;

  @EJB
  BarcodeScanner barcodeScanner;

  @PostConstruct
  public void init() throws IntegratedModelException {
    smr = new StateMachineRuntime<>(CashDeskStateMachine.class);
    smr.initialize();
    smr.initializeContainments();
    smr.initializeCrossReferences();
    cashDeskStateMachine = (CashDeskStateMachine) smr.getInstance();
  }

  @Operations
  public void addItemToCart() throws IntegratedModelException {
    items.add(barcodeScanner.scanItem());
    smr.executeTransition("scanCode");
  }

  @Operations
  public void checkout() throws IntegratedModelException {
    items.clear(); // Execute a real sale
    smr.executeTransition("finishSale");
  }
}
```

Listing 9.33: *CashDesk* implementation that uses the State Machine runtime

9.3.6 Implementing new Integration Mechanisms

When new integration mechanisms are developed, the code generation tool can be extended accordingly. The following process should be used to include new integration mechanisms into the code generation tool: First, the mechanism should be defined. A mechanism must include a unique name, a (possibly empty) meta model notation definition, and a model notation definition. Second a generic transformation should be defined (see Section 9.3.4), that can handle every instance of the mechanism. As the next step, a generic runtime should be defined (see Section 9.3.5). Then a type must be implemented that extends the abstract type `Mechanism` (see Section 9.2.3). The type must then implement three methods: The method `createMetaModelLibrary` generates the code structures, that represent the meta model element (see Section 9.3.3). The method `createTransformation` generates the code for the model-code transformation. The generated code should extend the generic transformation defined earlier. The method `createRuntime` generates the code for the specific runtime. The specific runtime should extend the generic runtime defined earlier. At last the new subtype of `Mechanism` must be included in the code generation tool, by adding it to the classpath and to the list of

mechanisms in the user interface.

9.3.7 Integrating Generated Code from Integration Mechanisms with Codeling

Codeling can be used to execute the generated transformations of the code generation tool. For transformations to be available within Codeling, the generated meta model library plugin (see Section 9.3.3) and the generated transformation plugin (see Section 9.3.4) must be executed in an Eclipse installation. The transformation plugin needs to specify an extension of the extension point `org.codeling.languageRegistry`, which declares a language definition type. The code generation tool generates such extensions and language definition types. To use these plugins in Codeling, the resulting projects must be made available to a Codeling installation, e.g. by exporting them as JAR files and placing them in the *dropins* folder of the Eclipse installation.

For a selection of integration mechanisms, code generators for meta model notations, transformations, and runtime stubs have been developed in the context of this thesis. These generators and the code generation tool are available on the data medium attached to this thesis (see Appendix B).

9.4 Strategy for Developing Transformations

The development of transformations for architecture implementation and specification languages is not an easy task, despite the support from Codeling and the code generation tool. A major challenge is to relate the elements of an architecture implementation or specification language to architectural concepts. This relationship is not always clear. Consider a program that comprises multiple projects in the Eclipse IDE. Each of these projects contains multiple program code structures that represent component types. What concept do the projects correspond to? They could represent deployment fragments, because often programs are splitted into different projects for defining deployment fragments. They could also represent composite component types in a hierarchical component type structure, or namespaces. Even when the same architecture language is used for multiple programs, such a relationship can differ between the different programs, depending on the preferences of the model's stakeholders. We propose the following strategy for developing transformations between architecture implementation or specification languages and the IAL.

Translations should be developed in a hierarchical manner (see Figure 9.29): for each architecture implementation or specification language, a **generic transformation** should exist, that can serve as basis for all other transformations. This generic transformation includes mostly undisputed translations. In these transformations disputable translations are either not implemented (e.g. a project in the IDE is not translated at all), or any default is used (e.g. a project is translated into a deployment fragment in the IAL).

Based on the generic transformations, **program-specific** transformations can be derived by tailoring, extension, and adaptation. In these translations, disputable relationships can be developed as it is required in the specific program.

Some languages are used for describing software architectures differently in the context of multiple groups. An example for such a broadly used language is the Unified Modeling Language (UML) [Obj15]. The UML is very flexible to use, and therefore groups of language users have developed certain habits for expressing designs and architectures using this language. Example for such groups are developers within a specific application or technical domain, developers

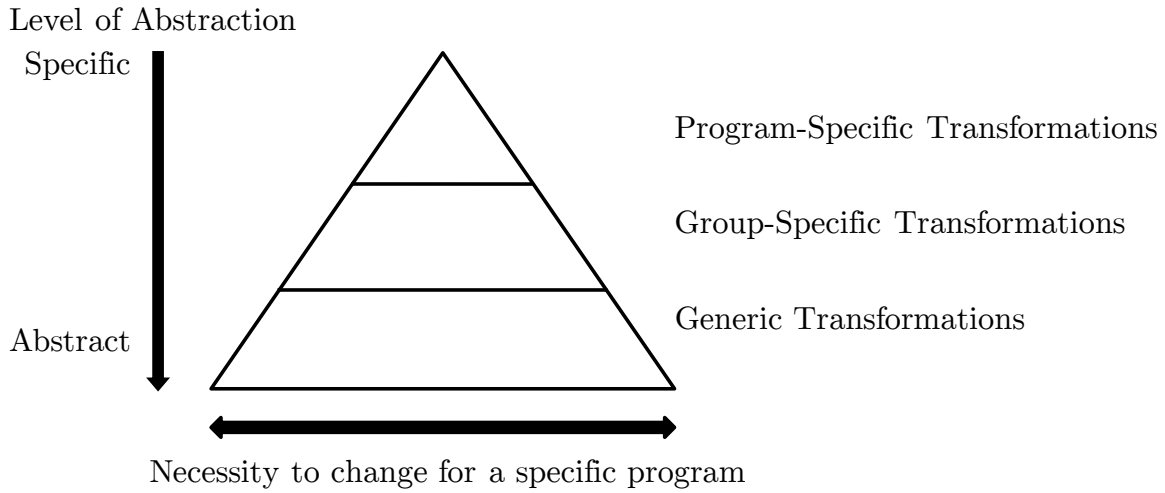


Figure 9.29: A hierarchical structure of transformation definition for architecture languages

within a specific industrial sector, or developers within specific groups of an organization. When translations are developed and reused within such groups, **group-specific translations** can be specified. Such translations contain more specific rules than generic transformations, but are still subject to tailoring, extension, and adaptation for specific programs within the group. All these transformations should be stored within a repository to enable reuse and adaptation.

9.5 Summary

This chapter described the implementation of the Explicitly Integrated Architecture Process, and a code generation tool for generating meta model notation code structures, model notation transformations, and runtime stubs for arbitrary meta models, following the Model Integration Concept.

Codeling implements the process, triggers the transformations, and executes inter-profile transformations. It also provides support for implementing transformations of the Model Integration Concept or architecture model transformations via libraries. The tools therefore provide a broad support for executing the Explicitly Integrated Architecture Process and for developing translations for further languages.

Part III

Evaluation and Conclusion

10 Evaluation

The objective of the evaluation is to find out whether the presented approach achieves the goal and requirements stated in Section 1.6. It has to be shown, that the approach:

R1 bridges the gap between architecture specification languages and implementations thereof,

R2 takes the difference of architecture specification and implementation languages into account.

R3 provides a single source of information for architecture descriptions,

R4 creates bidirectional translations between the architecture specification and implementation,

R5 is be prepared for architecture specification and implementation language emergence and evolution, and

This is achieved by evaluating whether the corresponding questions (see Section 1.6) can be answered with *yes*. For showing that the approach meets the requirement **R1**, it has to be implemented and applied to a case study that allows to integrate and execute non-architectural code in architectural code elements. To show that the approach handles the differences between architecture specification and implementation languages (**R2**), it has to be applied to a case study with two languages that have mutually exclusive language features. For showing that the approach meets the requirement **R3**, it can be shown in a case study that only the code view is required to derive architecture specification views. The fulfillment of **R4** can be shown by arguing about the bidirectionality of the transformations. To show the applicability of the approach with different languages, the approach must be applied to a second case study with different languages. The handling of emergent and evolving architecture languages (**R5**) can be shown by executing case studies with the same program as origin and different target languages.

First, the approach is applied to a total of four case studies. In the first case study (see Section 10.1), the approach is applied to JACK 3, an e-assessment tool that is implemented with the Java Enterprise Edition (JEE) 7. The case study translates a subset of EJB 3.2, CDI 1.2, and JSF 2.2 into a subset of the UML and back. It therefore bridges the gap between architecture specification languages and implementations thereof *(R1)*. The use case adds time resource demand information to operations in the specification model. The architecture implementation language cannot express this information. This case study shows that such differences are taken into account by the approach *(R2)*. The program code is the single, original source of information for architecture information. The specification model is derived from the program code *(R3)*. In this case study bidirectional transformations are developed *(R4)*. Finally, the case study translates between the JEE and UML. Together with the other case studies, this contributes to showing that the approach is prepared for language emergence and evolution *(R5)*.

In the second case study (see Section 10.2), it is applied to the Common Component Modeling Example (CoCoME) [HKW+08]. In this case study, the architecture implementation is a self-implemented Java-based architecture implementation language and is translated into a subset

Section	Case Study / Description	Addressed Requirements
10.1	Case Study: JACK 3	**R1**, **R2**, **R3**, contributes to **R4** and **R5**
10.2	Case Study: Common Component Modeling Example	**R1**, **R2**, **R3**, contributes to **R5**
10.3	Case Study: Specification Language Migration	**R1**, **R2**, **R3**, contributes to **R5**
10.4	Case Study: Implementation Language Migration	**R1**, **R2**, **R3**, contributes to **R5**
10.5	The Bidirectionality of Transformations	**R4**
10.6	Resource Demand	-

Table 10.1: An overview of the evaluation activities and which requirements they address

of the PCM. As with JACK, this case study bridges the gap between architecture specification languages and implementations thereof *(R1)*. It also translates between languages with different architectural aspects *(R2)*. The program code is the single origin source of information *(R3)*. Together with the other case studies, this case study contributes to showing that the approach is prepared for language emergence and evolution by translating between a different pair of languages *(R5)*.

The third case study (see Section 10.3) translates the CoCoME system into the UML as another architecture specification language for showing the handling of specification language evolution, i.e. the change of the specification language. The fourth case study (see Section 10.4) translates the CoCoME system into JEE as another architecture implementation language, for showing the handling of architecture implementation language evolution. Besides bridging the gap *(R1)* and translating between languages with different architectural aspects *(R2)*, the case studies rely on the program code as single source of architectural information *(R3)*. These case studies also contribute to showing that the approach is prepared for language emergence and evolution by translating between different pairs of languages *(R5)*.

Section 10.5 argues about the bidirectionality of the transformations within the approach *(R4)*. Section 10.6 shows limitations of the prototype tool regarding resource demand. The evaluation is discussed in Section 10.7. Table 10.1 gives an overview of the case study and argumentation and which requirement they address.

10.1 Case Study: JACK 3

In the first case study, the development of the e-assessment tool *JACK 3* is supported by generating an architectural view in the UML specification language. JACK 3 is the designated successor of an e-assessment tool (JACK 2 [Str16]) developed at the working group "Specification of Software Systems" (S3) of the institute paluno of the University of Duisburg-Essen, Germany. Its predecessor is used in the teaching and assessment of various disciplines, including programming, mathematics, and micro-economics. JACK 3 comprises two parts: a back end written in Java using the Eclipse platform as architecture implementation language, and a front end written in Java, based on the Java Enterprise Edition 7. The front end defines a user interface, data definitions, and business logic for e-assessments. The following interactions with the front end can be seen as major interactions: Teachers define courses in which students can or must solve exercises. Teachers provide standard solutions and feedback for possible errors. Students see their individually available courses. They create solutions for the exercises within courses. The presentation of results depends on the execution mode. When JACK is in the learning mode, the students can see feedback regarding their solution. In the assessment mode the students do not get such feedback. Instead the teachers can see the feedback and a

grading. The back end evaluates solutions against the defined standard solution using possibly various checkers, e.g. in programming exercises the code can be checked statically, e.g. for the existence or non-existence of specific structures; or dynamically, e.g. by executing the program with some specific input and checking the results. For other types of exercises, other types of checkers exist. The back end of JACK 2 is not changed during for development of JACK 3. Therefore this case study focuses on the front end for supporting the development of JACK 3.

In the context of this case study, a subset of the Java Enterprise Edition is considered an architecture implementation language. It is translated into a subset of the UML, containing a composite structure diagram. Figure 10.1 shows the architecture, as it is extracted in this case study. The resulting UML model is changed by adding, changing, and deleting elements. Then the code is changed according to the changes in the UML model automatically by Codeling.

10.1.1 Java Enterprise Edition in JACK 3

Java Enterprise Edition (JEE) 7 [Ora13b] is an umbrella standard for various technologies (see Section 2.3.5). This case study is focused on subsets of EJB 3.2, CDI 1.2, and JSF 2.2. The JACK 3 front end consists of three projects within the eclipse IDE. The project *jack3-core* contains data types and low level components for data handling. The project *jack3-business* contains higher level components for more complex tasks triggered by users. The project *jack3-webclient* contains view components for a web application.

Enterprise JavaBeans

Enterprise JavaBeans (EJB) 3.2 [EJB13] specifies the definition of business logic components called *Enterprise Beans*. Enterprise Beans are business logic components. Their execution runtime ("container") handle their life cycles and references between enterprise beans, including transactions and concurrency during interactions with them (amongst other features). Two types of enterprise beans exist: *Message-Driven Beans* and *Session Beans*. Message-driven beans are event-based components. As they are not used in the JACK 3 front end at the considered state of development, message-driven beans are out of the scope of the case study. Session Beans are operation based components, which provide an operation-based interface to their context, either explicitly as Java interfaces or implicitly all of their public operations. Three types of Session Beans exist: *Singleton Beans* are instantiated once by the EJB container. When an instance of a singleton bean is requested, each requester will get the same instance. *Stateful Beans* are instantiated on a per-session basis. A requester can create a session within the session bean container. During a session, each request by the session owner for a stateful bean will return its session specific instance. They are usually used to store session-specific data. *Stateless Beans* are business logic components that are not supposed to store any state (although it is technically possible, e.g. to create caches). The container creates a set of instances of stateless beans in a pool. The amount of existing instances of a stateless bean is typically based on the frequency of requests. For each request another instance can be returned.

Technically, session beans are annotated Java types with attributes and interfaces. The respective annotations can be found in the package `javax.ejb`. Listing 10.1 exemplarily shows two session beans as they can found in the JACK 3 program code. The listing in this section are shortened due to readability reasons. The type `UserService` is a stateless session bean, as declared by its annotation. Its supertype `AbstractServiceBean` provides operations and fields

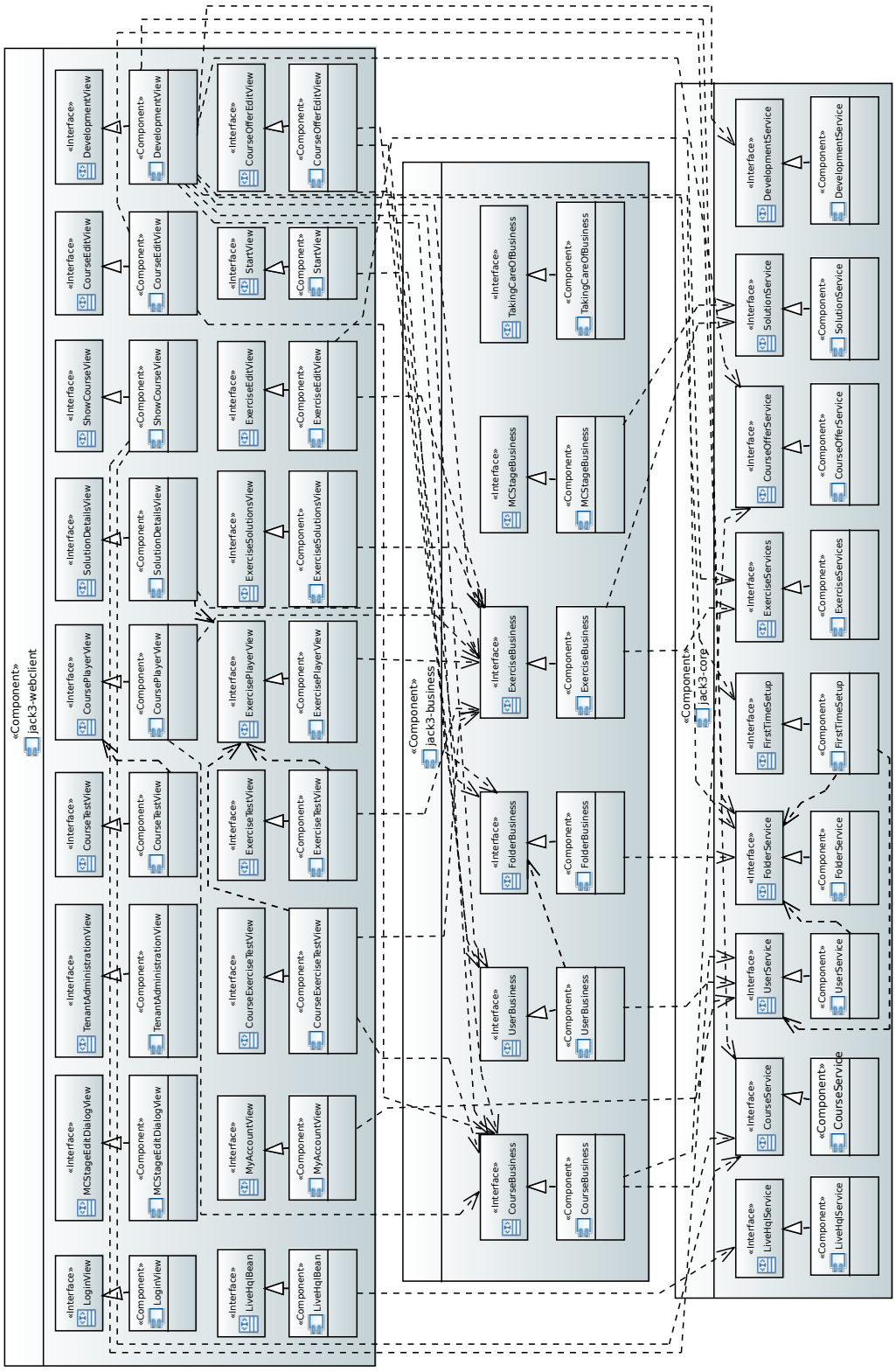


Figure 10.1: The UML architecture of JACK 3 in an UML view, as it is extracted in the case study. The diagram's layout has been set manually.

for a connection to databases and for logging. The `UserService` provides low level operations for managing users in the database. As an example, the public method `hasNoUser` returns true iff the database has no registered JACK users. The type `FirstTimeSetup` is a singleton session bean, as declared by its annotation. The additional annotation `@Startup` declares that the singleton instance is created by the container as soon as the bean is deployed. Else it would be instantiated lazily on a first request to the bean. The `FirstTimeSetup` bean has a reference to the `UserService`. The annotation `EJB` attached to the field `userService` orders the execution container to inject an instance of the stateless bean `UserService` into an instance of the bean `FirstTimeSetup`.

```
@Stateless                                                                    1
public class UserService extends AbstractServiceBean {                       2
  public boolean hasNoUser() {                                                3
    return countUser() < 1;                                                   4
  }                                                                           5
                                                                              6
  public User createUser(String screenName, String password, String email, boolean    7
      hasAdminRights,
      boolean hasEditRights) {                                                8
     //... Creates a user                                                     9
  }                                                                           10
  //... Further code for low level user management                            11
}                                                                             12
                                                                              13
@Singleton                                                                    14
@Startup                                                                      15
public class FirstTimeSetup extends AbstractServiceBean {                    16
  @EJB                                                                        17
  private UserService userService;                                            18
                                                                              19
  private void createTenantAdminUser() {                                      20
    if (userService.hasNoUser()) {                                           21
      userService.createUser(TENANT_ADMIN_NAME, null, null, true, false);     22
      getLogger().infof("Created␣Tenant␣Admin␣User␣\"%s\".",TENANT_ADMIN_NAME);    23
    }                                                                         24
  }                                                                           25
  //... Further code for the setup at startup                                 26
}                                                                             27
```

Listing 10.1: Exemplary session beans from the JACK 3 program code

Additionally, EJB defines data types components called *Entity Beans*. Entity beans allow for defining data types, which can be stored in relational databases using object-relational mappers [Amb03, Chapter 14]. Each data type represents a table in a relational database. Attributes are translated to database columns. References between entities are represented as columns or join tables, depending on their cardinality. Listing 10.2 shows an exemplary entity from the JACK 3 program code. The `User` is a data type as declared by the annotation `Entity`. The optional annotation `Table` defines the table name in the database that will hold the data. It declares a String attribute `screenName` (declared by the annotation `Column`), and a reference to another entity bean `ContentFolder` (not shown), which represent folders for user generated content, such as courses for teachers. The annotation `OneToOne` declares a reference of the said cardinality.

```
@Entity                                                                          1
@Table(name = "Usertable")                                                       2
public class User extends AbstractEntity implements Comparable<User> {          3
  @Column(nullable = false, unique = true)                                       4
  private String screenName;                                                     5
                                                                                 6
  @OneToOne                                                                      7
  private ContentFolder personalFolder;                                          8
  //... Constructors, Getters, and Setters                                       9
}                                                                                10
```

Listing 10.2: Exemplary EJB entity beans from the JACK 3 program code

Context and Dependency Injection

Context and Dependency Injection (CDI) 1.2 [JSR14] is a framework for specifying object life cycles within different scopes. In the context of JACK 3, the functionality is used to define business logic components called *Beans*. For a better differentiation between enterprise bean of EJB and beans of CDI they are called *CDI Beans* in the context of this thesis. Analogously to enterprise beans in EJB, CDI beans provide interfaces either explicitly as Java interfaces or implicitly by their public operations. In contrast to EJB, CDI beans do not automatically handle transactions or manage concurrent invocations of their instances. CDI beans have different scopes. The scopes in CDI are *request scoped*, *session scoped*, *application scoped*, or *conversation scoped*. Request scoped CDI beans are instantiated for each request. Session scoped CDI beans live for the length of a session, analogously to stateful beans of EJB. Application scoped CDI beans are instantiated exactly once during the application life cycle. Conversation scoped beans can be instantiated for manually defined conversations within a session or among multiple sessions.

CDI beans are implemented using annotated types with annotations from the package `javax.enterprise.context`. Listing 10.3 exemplarily shows two CDI beans as they can found in the JACK 3 program code. The type `UserBusiness` is a request scoped CDI bean, as declared by its annotation. Its supertype `AbstractBusiness` provides operations and fields for logging. The `UserBusiness` has a reference to the `UserService` shown in Listing 10.1 via a field annotated with `Inject`. The annotation triggers the CDI container to inject a suitable instance of the requested type. As CDI and EJB are highly integrated, the container will inject an instance out of the pool of instances of the stateless bean. It is possible to reference an EJB session bean from a CDI bean using the `Inject` or the `EJB` annotation. CDI beans can only be referenced via the `Inject` annotation. The `UserBusiness` provides business level operations for user management. As an example, the public method `createUser` creates a user using the injected `UserService`, and adjacent structures. The type `UserSession` is a session scoped CDI bean, as declared by its annotation. The additional annotation `Named` declares that instances of this bean can also be retrieved using a name-based registry. The `UserSession` bean also has a reference to the `UserService` shown in Listing 10.1. It provides operations for logging in and out, and for getting information about the currently logged in user. For these requests, it uses the `UserService`.

```
@RequestScoped                                                                    1
public class UserBusiness extends AbstractBusiness {                             2
  @Inject                                                                         3
  private UserService userService;                                                4
                                                                                  5
  public User createUser(String screenName, String password, String email, boolean  6
      hasAdminRights, boolean hasEditRights) {
    User user = userService.createUser(screenName, password, email, hasAdminRights,  7
        hasEditRights);
    return createPersonalFolderIfRequired(user);                                  8
  }                                                                               9
  //... Further code for business operations regarding users                      10
}                                                                                 11
                                                                                  12
@SessionScoped                                                                    13
@Named                                                                            14
public class UserSession implements Serializable {                               15
  @Inject                                                                         16
  private UserService userService;                                                17
  private User currentUser;                                                       18
                                                                                  19
  public String login() { [...] }                                                 20
                                                                                  21
  public String logout() { [...] }                                                22
                                                                                  23
  public User getCurrentUser() {                                                  24
    return currentUser;                                                           25
  }                                                                               26
  //... Further code for handling session data, getters and setters               27
}                                                                                 28
```

Listing 10.3: Exemplary CDI beans from the JACK 3 program code

JavaServer Faces

JavaServer Faces (JSF) 2.2 [Ora13a] specifies the definition of FacesBeans as view components for web pages. In contrast to EJB and CDI, it is coupled with a web framework. Faces Beans have different scopes. The scopes in JSF are *request scoped*, *session scoped*, *application scoped*, and *view scoped*. Request scoped beans, session scoped beans and application scoped beans work analogously to CDI beans of the corresponding scopes. View scoped beans are instantiated for the interaction with one specific web page. I.e. when a web page is opened, a new instance is created for the user. When from this web page a request is triggered to the same page, e.g. by submitting a form or executing asynchronous requests within the web page, the same instance will be used, as long as no other page is requested.

JSF beans are implemented by using annotations from the package `javax.faces.view`. Listing 10.4 exemplarily shows a faces bean which can be found in the JACK 3 program code. The abstract type `AbstractView` is an abstract class used as the basis for multiple faces beans. It provides the operation `getCurrentUser` to get the user of the current session. It uses the session scoped CDI bean `UserSession` in this operation, which is shown in Listing 10.3. The type `MyAccountView` is a view scoped Faces Bean, as declared by its annotation, for a web page that allows for viewing and changing user related data in JACK. It extends `AbstractView` and uses its operation `getCurrentUser` to populate the view.

```
public class AbstractView {
  @Inject
  private UserSession userSession;

  public User getCurrentUser() {
    return userSession.getCurrentUser();
  }
  //... Further supplemental code for faces beans
}

@ViewScoped
@Named
public class MyAccountView extends AbstractView implements Serializable {
  private String screenName;
  private String email;
  //... Further fields

  public void loadMyAccount(){
    screenName=getCurrentUser().getScreenName();
    email=getCurrentUser().getEmail();
  }
  //... Further code for populating the view
}
```

Listing 10.4: Exemplary faces bean from the JACK 3 program code

10.1.2 Java Enterprise Edition Meta Model

The JEE 7 meta model for the case study is built based on the corresponding specifications. Figure 10.2 shows the meta model described with Ecore. The root element is the *Architecture*, which contains archives, namespaces, beans, and entities. An *archive* corresponds to a project within the IDE. It references the entities and beans of the architecture. A *namespace* represents a package in Java. Namespaces reference beans and entities. They are hierarchically structured. Session bean, CDI beans, and Faces Beans have boolean attributes that denote their scopes. Exactly one of these attributes must be true. All beans are named, may have operations, and may reference each other. These common features are captured in the abstract class *Bean*. *Entities* own *entity attributes*, which have a type and a cardinality. Entities may also reference each other. Both beans and entities may have *operations*. These have *operation parameters* as parameters or return types. Operation parameters also have a cardinality and a type. The type may be either primitive, represented by the attribute *primitiveType*, or an entity, represented by the reference *entityType*. At most one kind of type must be set. If no type is set of a return type parameter, *Void* is assumed.

10.1.3 Unified Modeling Language in JACK 3

The Unified Modeling Language (UML) [Obj15] is a language for describing systems, especially software systems and their context. A meta model for UML 2.5 has been developed in the context of Eclipse's Model Development Tools initiative[1]. During interviews with the JACK 3 development lead, the requirement arose to have an architectural view on JACK 3 in a

[1]Eclipse Model Development Tools – `https://www.eclipse.org/modeling/mdt/?project=uml2`

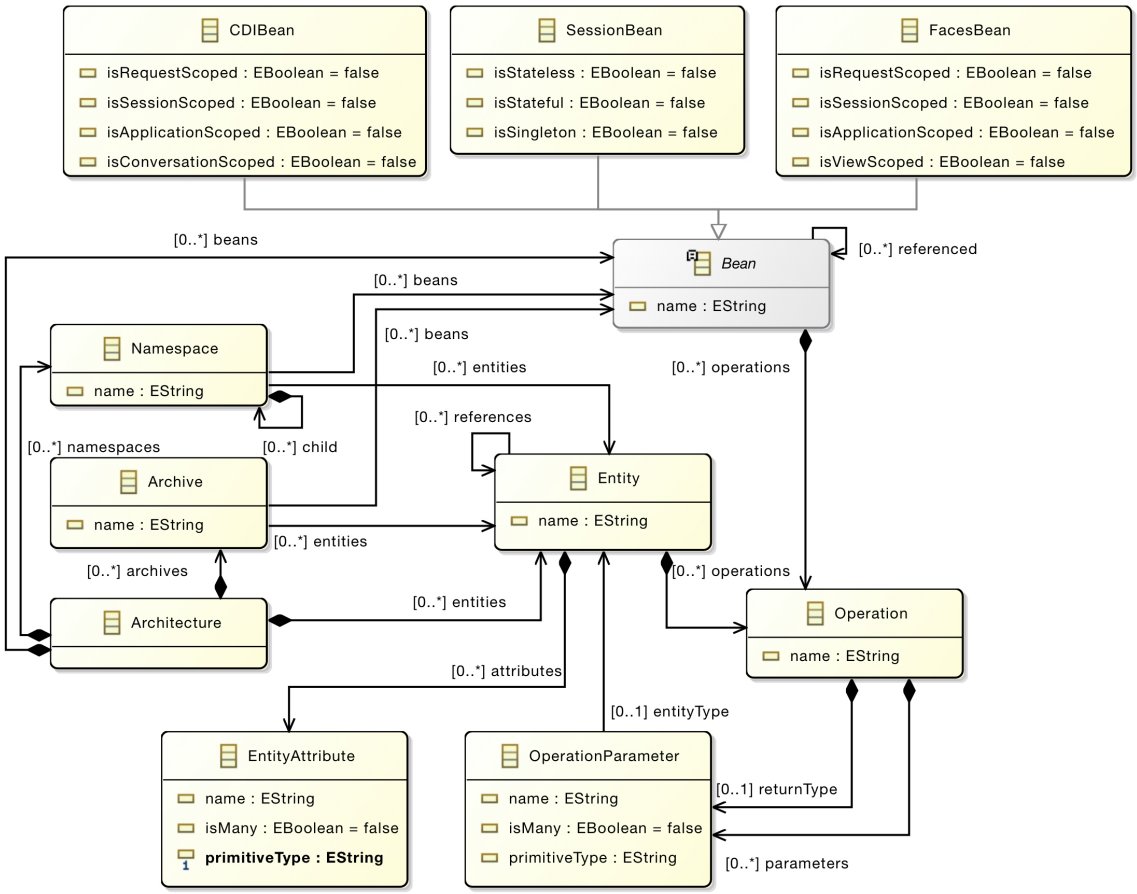


Figure 10.2: The JEE 7 meta model of the JACK case study

UML composite structure diagram, with the projects in the IDE as composite components, which have the beans as child components. This view should produce a visual feedback on how the components are spread throughout the projects, and how they are interconnected. Therefore in the context of this case study, components, interfaces, operations, and component interconnection are used to represent the JACK 3 architecture.

10.1.4 Model Integration Concept

In this case study the architecture implementation language's meta model was derived from a specification. The JEE specification defines program code structures for representing architectural elements, for which notations (see Definition 33) had to be created. For some of these notations, integration mechanisms could be used or adapted to the JEE specification. Table 10.2 shows the mapping between meta model elements and integration mechanisms, or a short description of the notation respectively. As an example, Listing 10.5 shows the implementation of the translation between the program code and a CDI bean in the JEE meta model. It extends the `TypeAnnotationTransformation` shown in Listing 9.15 and changes the expected annotation to one of the CDI bean annotations from the specification. When new model elements are to be created in the program code, a request scoped CDI bean will be created. The constructor configures the abstract supertype with the model class, that is subject to translation, and with the transformation of the owning reference, in this case a translation for an *Architecture* element. The `doCreate*` operations create child transformations for attributes and references of the bean. For the abstract class *Bean* in the meta model, a transformation helper type `BeanTransformation` exists, that creates child transformations for the references *referenced* and *operations*.

For the architecture, namespaces, entities, beans, and operations, integration mechanisms can be used or adapted by changing the required annotations in annotation-based mechanisms. It is not necessary to generate meta model notations as described in Section 9.3.3, because corresponding annotations already exists in the JEE API [Ora13b].

10.1.5 Architecture Model Transformations

Architecture model transformations have been implemented in this case study between the JEE meta model and the IAL, and between the IAL and the UML.

Java Enterprise Edition

A TGG was developed to translate between the JEE meta model and the Intermediate Architecture Description Language meta model (see Chapter 6). The complete TGG comprises 35 rules and can be found on the data medium associated with this thesis (see Appendix B).

The general scheme of the translation is shown here with a small example. Figure 10.3 shows the TGG rule for translating between an architecture element of the JEE meta model and an architecture element of the IAL with corresponding stereotypes. Figure 10.4 shows the TGG rule for translating between a stateful EJB session bean element of the JEE meta model and a component type with a provided interface and a component instance representative of the IAL with corresponding stereotypes. A stateful EJB session bean is mapped to a component type, that provides an interface. EJB allows beans to have an implicit interface. This is

```
package org.codeling.lang.jee7.transformation;
// ... imports
public class CDIBeanTransformation extends TypeAnnotationTransformation<CDIBean> {

  public CDIBeanTransformation(
      AbstractModelCodeTransformation<? extends EObject, ? extends IJavaElement>
          parentTransformation) {
    super(parentTransformation, JEE7Package.eINSTANCE.getCDIBean());
  }

  @Override
  public void doCreateCrossReferencesTransformations(
      List<AbstractModelCodeTransformation<? extends EObject, ? extends IJavaElement>>
          result) {
    new BeanTransformation().doCreateCrossReferencesTransformations(this, result);
  }

  @Override
  protected void doCreateChildTransformationsToCode(
      List<AbstractModelCodeTransformation<? extends EObject, ? extends IJavaElement>>
          result) {
    new BeanTransformation().doCreateChildTransformationsToCode(this, result);

    // Child transformations for attributes
    result.add(new IsRequestScopedTransformation(this));
    result.add(new IsSessionScopedTransformation(this));
    result.add(new IsApplicationScopedTransformation(this));
    result.add(new IsConversationScopedTransformation(this));
  }

  @Override
  protected void doCreateChildTransformationsToModel(
      List<AbstractModelCodeTransformation<? extends EObject, ? extends IJavaElement>>
          result) {
    new BeanTransformation().doCreateChildTransformationsToCode(this, result);

    // Child transformations for attributes
    result.add(new IsRequestScopedTransformation(this));
    result.add(new IsSessionScopedTransformation(this));
    result.add(new IsApplicationScopedTransformation(this));
    result.add(new IsConversationScopedTransformation(this));
  }

  @Override
  public boolean hasExpectedAnnotation(IType type) {
    return ASTUtils.hasAnnotation(type, "javax.enterprise.context.RequestScoped",
        "javax.enterprise.context.SessionScoped", "javax.enterprise.context.
            ApplicationScoped",
        "javax.enterprise.context.ConversationScoped");
  }

  @Override
  protected String getNewAnnotationName() {
    return "javax.enterprise.context.RequestScoped";
  }
}
```

Listing 10.5: The implementation of the translation between the JEE program code and a CDI bean in the JEE meta model

Meta Model Element	Integration Mechanism	Notation Details
Architecture	**Ninja Singleton**	
→ all references		Included in the owner's mechanism
Archive		**Project in the IDE**
→ beans		Bean definitions within the Project
→ entities		Entities definitions within the Project
Namespace	**Namespace Hierarchy**	
→ all references		Included in the owner's mechanism
CDIBean	**Type Annotation**	**Annotations from the specification**
→ all attributes		Identified by the existence of the corresponding annotations
SessionBean	**Type Annotation**	**Annotations from the specification**
→ all attributes		Identified by the existence of the corresponding annotations
FacesBean	**Type Annotation**	**Annotations from the specification**
→ all attributes		Identified by the existence of the corresponding annotations
Bean		**Not translated, because it is abstract**
→ referenced	Annotated Member Reference	Annotations `@EJB` or `@Inject`
→ operations	Containment Operation	No annotation required on operation
Entity	**Annotated Member Reference**	**Annotation `@Entity`**
→ references	Annotated Member Reference	Annotations from the specification
→ operations	Containment Operation	No annotation required on operation
EntityAttribute		**A field with the annotation `@Column`**
→ isMany		`true` iff the field is an array or collection type
→ primitiveType		The Java type of the field, if it is not of an entity type
Operation		**Derived from the operation owners**
→ parameters		Each parameter of the operation
→ returnType		The return type of the operation
OperationParameter		**Derived from the parameter owner**
→ isMany		`true` iff the field is an array or collection type
→ primitiveType		The Java type of the parameter, if it is not of an entity type
→ entityType		The Java type of the parameter, if it is not of a primitive type

Table 10.2: The mapping of meta model elements to notations in the JACK 3 case study

translated into a component type with a corresponding explicit interface element. Figure 10.5 shows the TGG rule for translating between a reference from an architecture to beans of the JEE meta model and a reference from an architecture to the corresponding component type with an associated provided interface and corresponding stereotypes in the IAL.

When translating architectures from the IAL to JEE, it is possible that the component types in the IAL does not have all required stereotypes, that are shown on the right side of Figure 10.3. In this case the rule would not be executed and no architecture would be translated. The profile activation rules (see Section 7.2.4) are used automatically in the Explicitly Integrated Architecture Process to ensure the existence of these stereotypes. For missing stereotypes on component types, default translation rules have been added to the TGG, so that new components can be added and translated into JEE compliant program code. Figure 10.6 shows the this translation rule for component types. The upper half of this figure shows a negative application condition, which declares that the given rule will only be executed when no part of

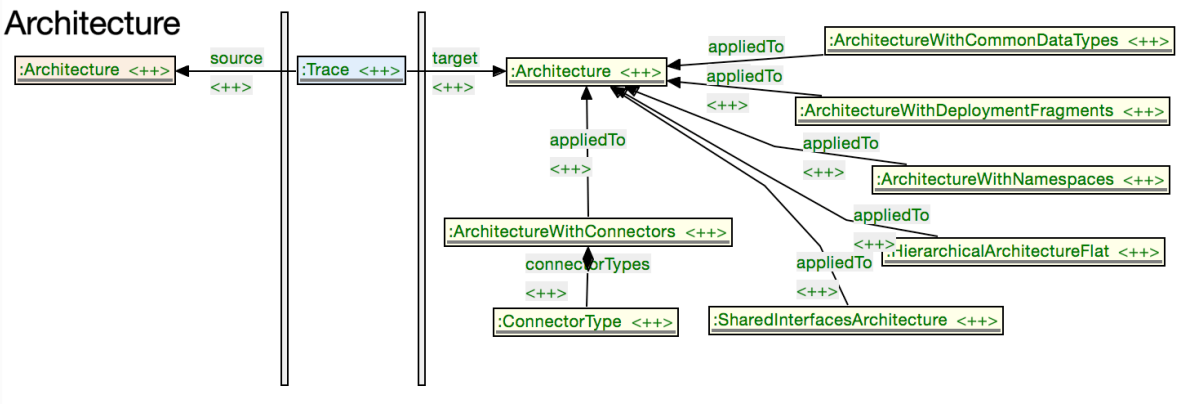


Figure 10.3: The TGG rule for JEE architectures in the JACK 3 case study

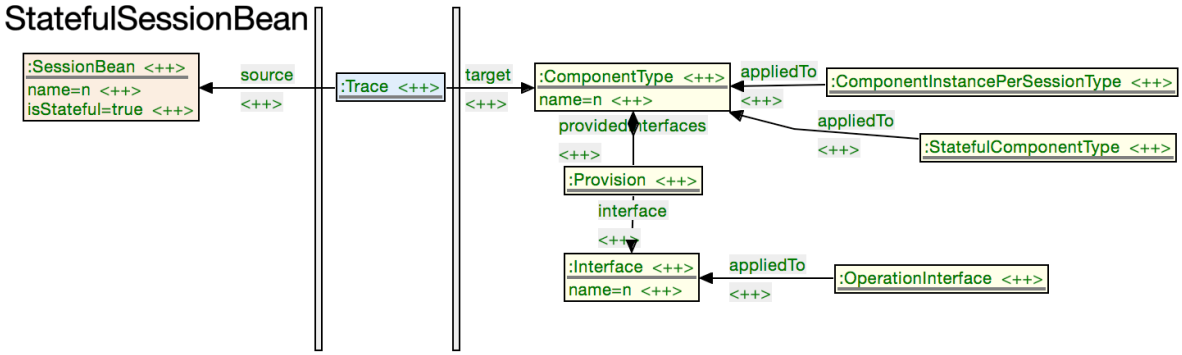


Figure 10.4: The TGG rule for JEE stateful beans in the JACK 3 case study

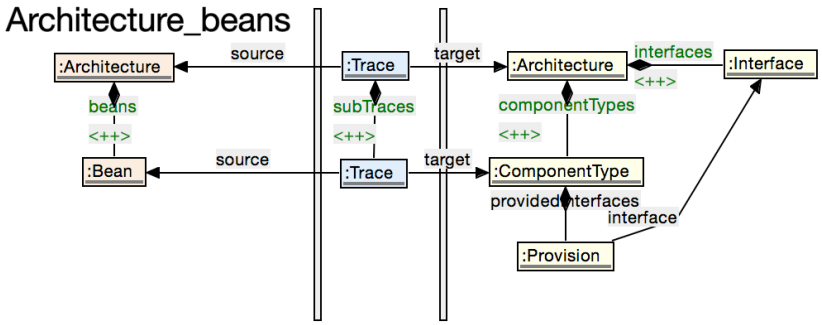


Figure 10.5: The TGG rule for the reference from JEE architectures to beans in the JACK 3 case study

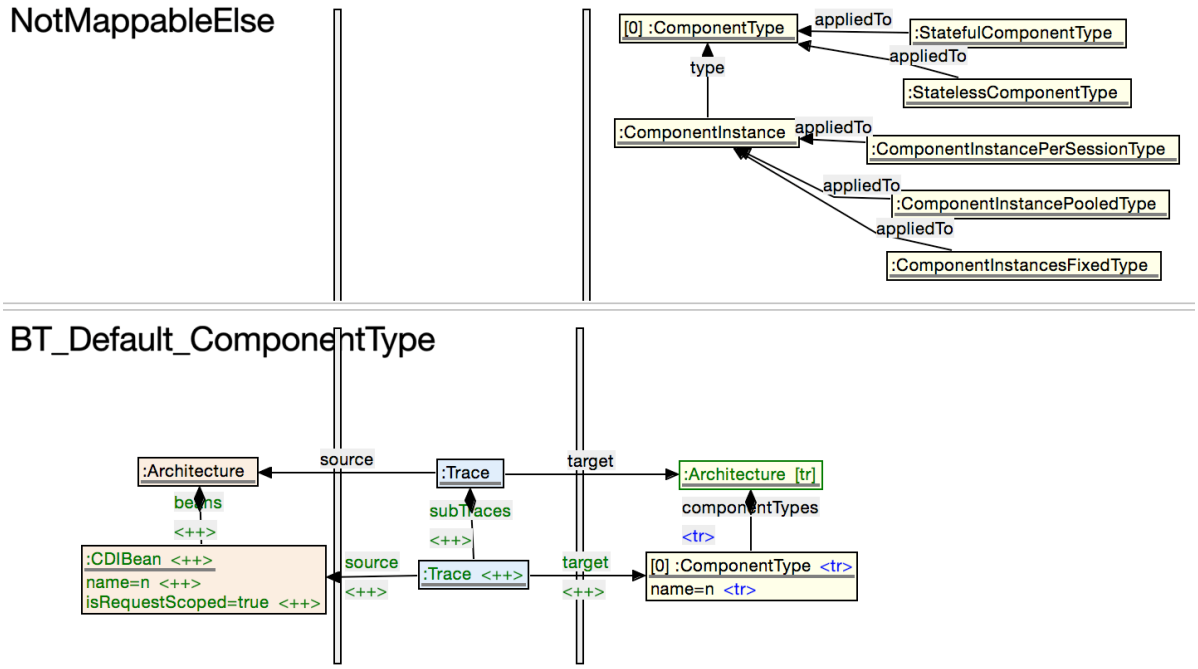


Figure 10.6: The TGG default rule for component types in JEE in the JACK 3 case study

the NAC can be matched. The *ComponentType* element in Figure 10.6 has the number 0. This number shows a mapping between this element and the *ComponentType* element in the NAC.

Unified Modeling Language

A TGG was developed to translate between the IAL (see Chapter 6) and the UML. The complete TGG comprises nine rules and can be found on the data medium associated with this thesis (see Appendix B).

The general scheme of the translation is shown here in a small example. Figure 10.7 shows the TGG rule for translating between an architecture element of the IAL meta model and a *Model* element of the UML. Figure 10.8 shows the TGG rule for translating between a deployment fragment of the IAL meta model and a component in the UML. The translation for subcomponents is shown in Figure 10.9.

The case study includes (a) the definition of a meta model of a subset of the JEE; (b) notations for the JEE meta model in Java, that comply to the JEE API, partly based on integration mechanisms; (c) architecture model transformations between the JEE meta model and the IAL, and (d) architecture model transformations between the IAL and a subset of the UML. In the case study, UML components, their interfaces, operations, and relations are extracted from the program code. Figure 10.1 shows the architecture as it is extracted in the case study. The operations are omitted for a better overview.

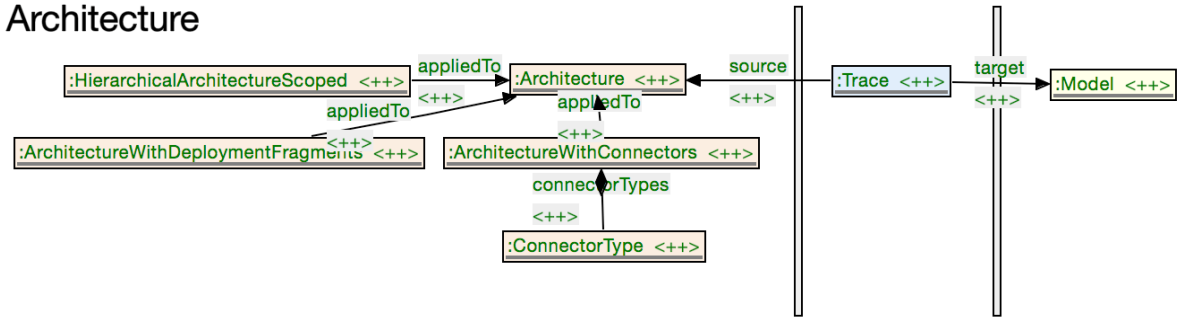


Figure 10.7: The TGG rule for UML models in the JACK 3 case study

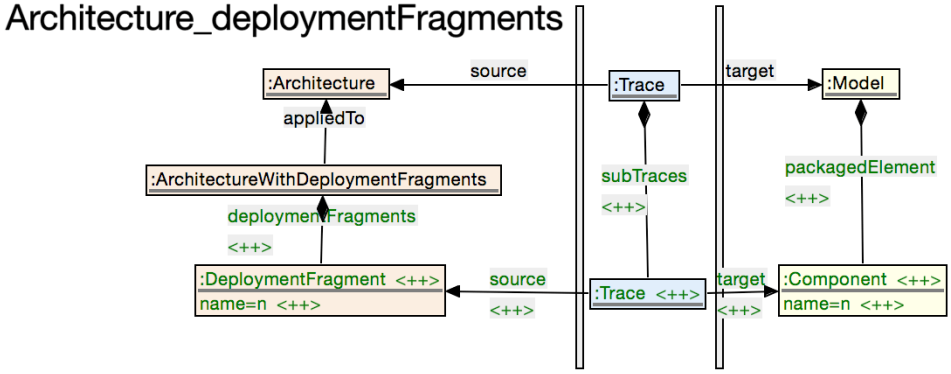


Figure 10.8: The TGG rule for UML deployments in the JACK 3 case study

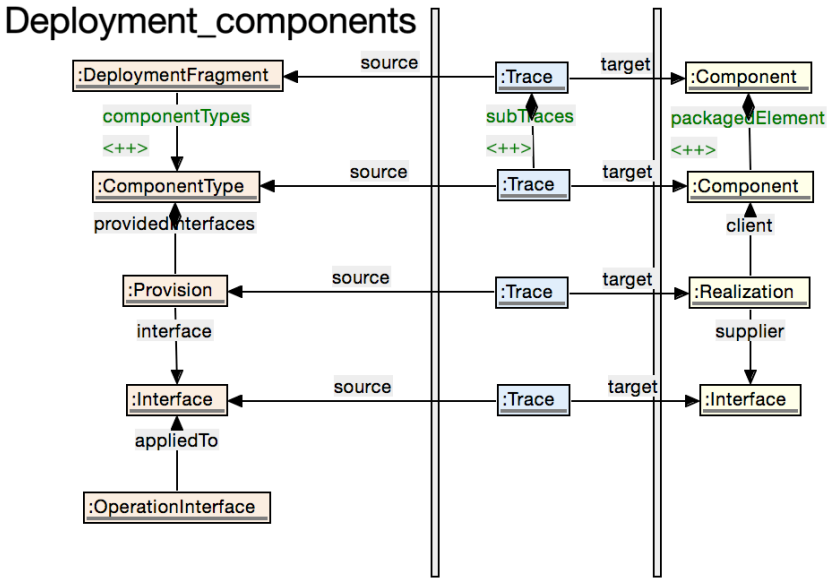


Figure 10.9: The TGG rule for components within deployment fragments in the JACK 3 case study

UML distinguishes between the model data and diagrams presenting model data. Codeling integrates model data with program code. It does not create UML diagrams with the corresponding layout information. The model elements can be viewed with a UML editor based on the Ecore UML meta model, such as Papyrus[2]. The important aspect to see in Figure 10.1 is that a layered architecture [GHJV95] is implemented. No component accesses a component in a layer above its own. However, many view components (in the upper layer) have a direct dependency to the core. An interview with the development leader has shown that this circumstance was known, and the diagram gave an overview, which component has to be revisited.

10.1.6 Model Changes

In the case study, four changes are made to the specification model. A new component type is added, the name of a component type is changed, one component type is deleted, and a time resource demand for an operation is added. First, a new component type *AdministrationService* is added to the composite component *jack3-core*, that realizes an interface with the same name as the component. On saving the changed model, Codeling translates this model change into code changes, resulting in a new Java type within the project *jack3-core* as shown in Listing 10.6. The new type is a request scoped CDI bean, following the translation for new components as shown in Figure 10.6. The package that owns the type is `new_architecture_elements`, a default package declared in the notations, because the architecture model did not associate any specific namespace to the bean.

In a second change the component type *MyAccountView* is renamed to *AccountDetailsView*.

[2]Papyrus – `https://eclipse.org/papyrus/`

```
package new_architecture_elements;

import javax.enterprise.context.RequestScoped;

@RequestScoped
public class AdministrationService {}
```

Listing 10.6: Newly created bean in the JACK 3 program code after changing the UML model

When the changes are propagated to the program code, this change results in a renaming refactoring on the Faces bean type `MyAccountView`. The refactoring is implemented in the method `updateCodeFragments` of the type `TypeAnnotationTransformation` shown in Listing 9.25.

The third change is the deletion of the component type *LoginView*, its interface with the same name, and the interface realization reference. When these changes are propagated to the code, the corresponding type declaration is deleted. The deletion is implemented in the method `deleteCodeFragments` of the type `TypeAnnotationTransformation` shown in Listing 9.25. In this specific scenario, no other component references the *LoginView* component. When a referenced component is deleted, the references to this component will also be deleted.

The fourth change is the addition of a time resource demand to the operation `getAllCoursesForUser` of the component `CourseService` in the layer *jack3-core*. To express the resource demand, in this case study a structured comment upon the operation with the content `Time Resource Demand: [50ms]` is used. In a more sophisticated UML model, a profile should be defined. When the change is propagated to the program code, a new annotation is added to the operation, which states the time resource demand as a string value. Listing 10.7 shows the annotation, which has been added to the operation.

```
@Stateless
public class CourseService extends AbstractServiceBean {

  [...]

  @TimeResourceDemand(duration="50ms")
  public List<Course> getAllCoursesForUser(User user) {
    [...]
  }
}
```

Listing 10.7: The operation `getAllCoursesForUser` in JACK 3 extended with a resource demand

The data medium attached to this thesis (see Section B) contains the artefacts of this case study: The program code and model, including all intermediate models, each before and after the model change, the architecture model transformations, and the transformation types for the notations of the Model Integration Concept.

10.2 Case Study: Common Component Modeling Example (CoCoME)

In the second case study, the program code of the Common Component Modeling Example (CoCoME) [HKW+08] is translated into a subset of the Palladio Component Model (PCM) [BKR09]. CoCoME has been developed as a benchmark for comparing software architecture languages. The original CoCoME benchmark artefacts provide the context, the

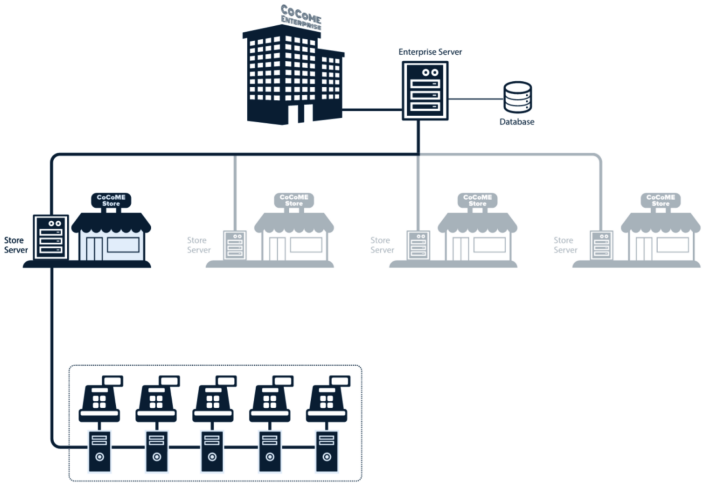


Figure 10.10: An overview of the CoCoME structure [HRR16]

requirements, the design, and the implementation of a system. The system drives the business for an enterprise that runs multiple stores. Each store contains multiple cash desks in a cash desk line [HKW+08]. It is implemented in plain Java without using a standardized architecture implementation language. CoCoME has recently been revisited [HRR16] in the context of the Priority Programme 1593 "Design For Future – Managed Software Evolution"[3] by the German Research Foundation (Deutsche Forschungsgemeinschaft (DFG)), where evolution scenarios for the CoCoME system have been developed as a platform for collaborative research. These scenarios include a migration to a JEE implementation, and the addition of a pick up store. In the context of this case study the original CoCoME implementation is used for evaluating the use of the Explicitly Integrated Architecture Process for software that does not use standardized frameworks, but create product-specific architectural abstractions. It is translated into an architectural structure expressed with a subset of the PCM as a preparation for a performance analysis.

10.2.1 The CoCoME System

Figure 10.10 gives an overview of the CoCoME system. CoCoME provides components that build a cash desk, including bar code scanners, cash boxes, and printers, and a cash desk line comprising multiple cash desks. A store has a cash desk line and a store server, which manages the stock and the payment. Multiple stores are managed by an enterprise. The enterprise server gives an overview about the sales and stock throughout the enterprise, provides metrics, algorithms for finding out whether a store needs to be refilled with stock short hand by another store nearby, and triggers for such a transport.

The original structural architecture of CoCoME is shown in Figure 10.11. The components within the cash desk line communicate with each other using an event bus. The cash desk line provides an outgoing connection via an operation-based banking interface for handling

[3]DFG Priority Programme 1593 – `http://dfg-spp1593.de`

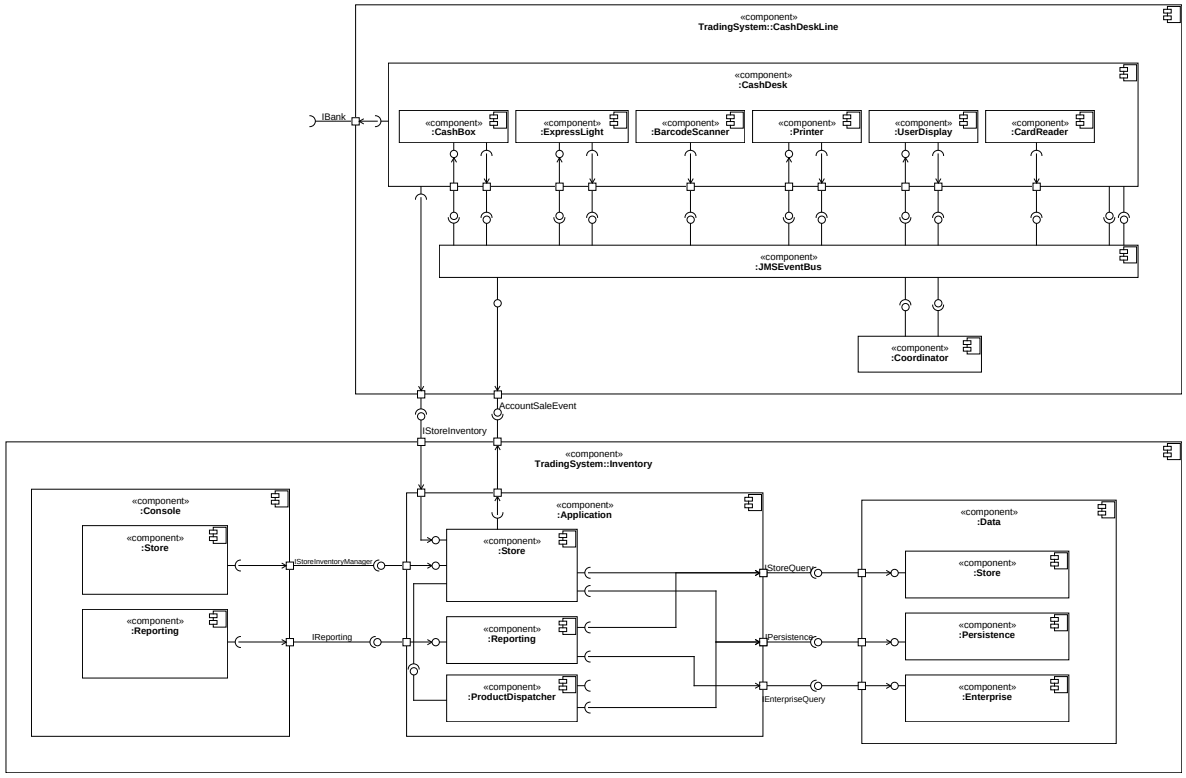


Figure 10.11: An overview of the CoCoME architecture in UML [HRR16]

payments. The inventory comprises the business logic for the reporting and controlling of the servers, and consoles for users to interact with the reporting and control mechanisms. The components within the application component execute the business logic using a data base connection realized by the data components. The components within the inventory communicate using operation-based interfaces. Between the cash desk line and the inventory, both events and operations are used for communication.

10.2.2 CoCoME Meta Model

The CoCoME meta model for the case study has been built based on a manual program code analysis. Figure 10.12 shows the meta model described with Ecore. The root element is the *Architecture*, which contains components. In the CoCoME program code a component is a (possibly empty) set of type, interface, and enumeration declarations within a namespace in Java. Components may define, dispatch, or handle *events*, require or provide operation-based *interfaces*, and may contain other components. Four types of components exist. *Servers* are stateless components that define *TransferObjects* as datatypes. They are active, meaning that they are executable using a *main* method in Java. Servers implement the business logic of CoCoME. They provide functionality for other components to use. *Consoles* are active, stateless components which provide UI elements to users. They use servers for executing their functionality. *Models* are stateful components. They are used to implement the cash desk line

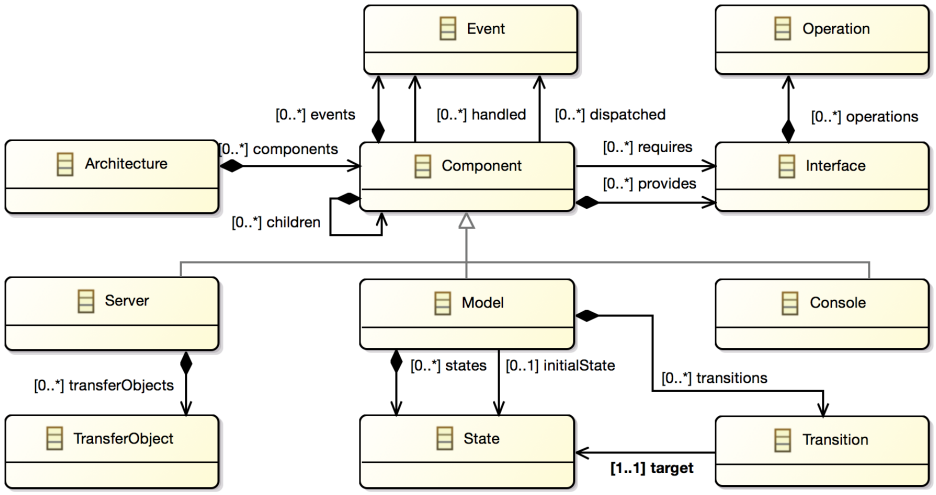


Figure 10.12: The AIL meta model of the CoCoME case study

and its subcomponents. Some model's behaviour can be described with a state machine, that comprises states and transitions from a set of possible states to a target state. *Components*, the superclass of all component types in the meta model, are instantiable themselves, but do not have an own functionality. They comprise child components, may delegate the provision or requirement of interfaces, and may own events. The figure of the meta model does not show the naming. The classes *Component*, *Interface*, *Operation*, *Event*, *State*, *Transition*, and *TransferObject* have an attribute *name* of the type String.

10.2.3 Model Integration Concept

The program code of CoCoME is not based on a standardized architecture implementation language. The component life-cycle and communication is implemented in plain Java. In contrast to the JACK 3 case study (see Section 10.1), the program code structures, that define architecture elements, such as components, cannot be derived from a specification. The program code had to be analyzed, and suitable program code structures had to be identified for this case study. Table 10.3 summarizes the identified structures.

Seven of 24 meta model elements (classes and references) could be translated by instantiating an integration mechanisms, some with small changes from the original mechanisms. The code generation tool (see Section 9.3) was used to generate the general structure for transformations. An empty mechanism *Custom Translation* has been used in the tool for creating empty transformation types for the meta model elements that follow none of the declared mechanisms. The tool generates a translation type for the meta model element (class, reference, containment reference, or attribute) mapped to that empty mechanism. The generated translation types provide a set of utility methods. For translations of references, they also ensure that target objects exist before references are translated. Using these structures, the effort for implementing the translations was reduced to implementing the methods `transformToModel` and to declare how the targets of model references are to be found within a code element. For notations

Meta Model Element	Notation Details
Architecture	The architecture is a root model element without a representation in the code. It follows the Ninja Singleton mechanism.
→ components	The derivation of the notation from the Ninja Singleton mechanism includes the translation of the containment reference *components*.
Event	Events are type declarations with names that end with `Event`.
Interface	A component interface is an interface within the package of a component, that is implemented by the component's main type. It follows the Static Interface mechanism, but uses no annotation.
→ *operations*	An interface owns an operation when it declares a corresponding method. This follows the Containment Operation mechanism without annotations.
Operation	The derivation of the notation for *Interface.operations* from the Containment Operation mechanism includes the translation of the class *components*.
Component	General components are represented by Java packages below the hierarchy `org.codeling.tradingsystem`. Their code structure is close to the Namespace Hierarchy mechanism, but also contains non-containment references to events and interfaces. Java packages only represent components when they have child components. Otherwise utility packages would be identified as components.
→ *children*	The derivation of the notation from the Namespace Hierarchy mechanism includes the translation of the containment reference *children*.
→ *dispatches*	The dispatching of events is not handled consistently in the CoCoME program code. In some cases helper methods are used for instantiating the event types and giving them to the event bus. In other cases no helper methods are used. Experiments have shown that the instantiation of an event type within a component's main type suffices to identify an event dispatch. However this translation is not bidirectional.
→ *handles*	A component handles events when its package contains an interface called `IComponentNameEventConsumer`, where `ComponentName` is the component's name. This interface declares one or more methods `onEvent`, that each take an event type as parameter. An `EventHandler` type implements this interface and calls the appropriate methods in the component's main type when these events occur.
→ *provides*	A component provides an interface by implementing it. This follows the Static Interface Implementation mechanism without an annotation.
→ *requires*	The requirement of interfaces is not consistently implemented. Required interfaces are given to a component via its constructor. Some interfaces are given as parameters. Other interfaces are type attributes of a helper class for distributed communication given as parameters to the constructor.
TransferObject	A transfer object is a type with a name ending on *TO* within a component's package.
Server	Server components are packages that own a type ending with `Server`. The package and all of its types belong to the component.
→ transferObjects	A server owns a transfer object when its package contains the corresponding type.
State	The implementation of state declarations is not consistent in CoCoME. In the cash desk model component, the states are enumeration items in an enumeration within an own compilation unit (a *.java* file) in a component's package. In other model components such enumerations are embedded into the component's main type. The program code has been adapted to follow the structure in the cash desk model component.
Model	Model components are packages that own a type ending with `Model`. The package and all of its types belong to the component.
→ *states*	With the program code adapted as described in the notation of states, a model component owns all states declared in its state enumeration.
→ *initialState*	The initial state is not implemented consistently in CoCoME. In general, the initial state is stored on the initialization of the model component's main type in a field named *state*. In the cash desk component this field has a constant value that declares the initial state. In other cases the field is written via the constructor or indirectly using methods with varying names or name structures. The program code has been adapted to follow the structure in the cash desk model component.
Transition	Transitions are not consistently implemented in CoCoME. A transition is identified by setting the state field to another value. In the cash desk model component this change happens directly in the business logic. In other state-based model components this is achieved using a setter method.
→ *target*	A transition's target is identified by the value that is set as described for transitions.
Console	Console components are packages that own a type ending with `Console`. The package and all of its types belong to the component.

Table 10.3: The mapping of meta model elements to notations in the CoCoME case study

based on integration mechanisms, these mechanisms had to be tailored. E.g. the operations of interfaces have no annotation, although the mechanism Containment Operation expects one.

Architecture Model Transformations for the CoCoME Meta Model

A TGG was developed to translate between the CoCoME meta model and the IAL. The complete TGG comprises 17 rules and can be found on the data medium attached to this thesis (see Appendix B). The general scheme of the translation follows the same patterns as those in the JACK case study between JEE and the IAL (see Section 10.1). The transformations between the CoCoME meta model and the IAL are not bidirectional. The case study implied the translation to PCM for preparing a simulation. Therefore bidirectionality was not enforced, although it is conceptually possible.

Architecture Model Transformations for the Palladio Component Model

A TGG was developed to translate between the IAL and a subset of the PCM. The complete TGG comprises 36 rules and can be found on the data medium associated with this thesis (see Appendix B). The general scheme of the translation follows the same patterns as those in the JACK case study between the IAL and the UML (see Section 10.1).

10.2.4 Model Simulation

The resulting model contains a PCM repository and a PCM system definition. Figure 10.13 gives an overview of the resulting PCM repository. Events are not included in the translation to PCM in this case study. For also translating events, it would be necessary to create more TGG rules. Figure 10.14 shows the derived PCM system, which, as an assumption, instantiates each topmost component exactly once.

For executing a simulation, the model needs to be extended with a resource environment model that contains the definition of connected hardware nodes with resource definitions, an assembly model, that assigns component instances to these hardware nodes, and a usage scenario model that describes an example usage scenario and a load definition.

For a meaningful simulation, PCM requires abstract behaviour specifications (Service Effect Specifications (SEFFs)) with resource demands. This information is not subject to the translation in the case study. In the case study, minimal SEFFs were created, that only include a start and an end node. Therefore abstract behaviour has to be manually added for a simulation.

10.3 Case Study: Specification Language Migration

When an architecture specification language evolves, or a new architecture specification language emerges, an existing software system can be migrated to another architecture specification language using Codeling. I.e. different views can be generated of the same program code, in different architecture specification languages. In the CoCoME case study above, a PCM specification was extracted. In this case study, a UML specification of the CoCoME system (see Section 10.2.1) is extracted. This is a case of architecture specification language migration. The step 1 (Program Code to Translation Model) of the Explicitly Integrated Architecture Process in this case study is equal to the corresponding step in the CoCoME to PCM case

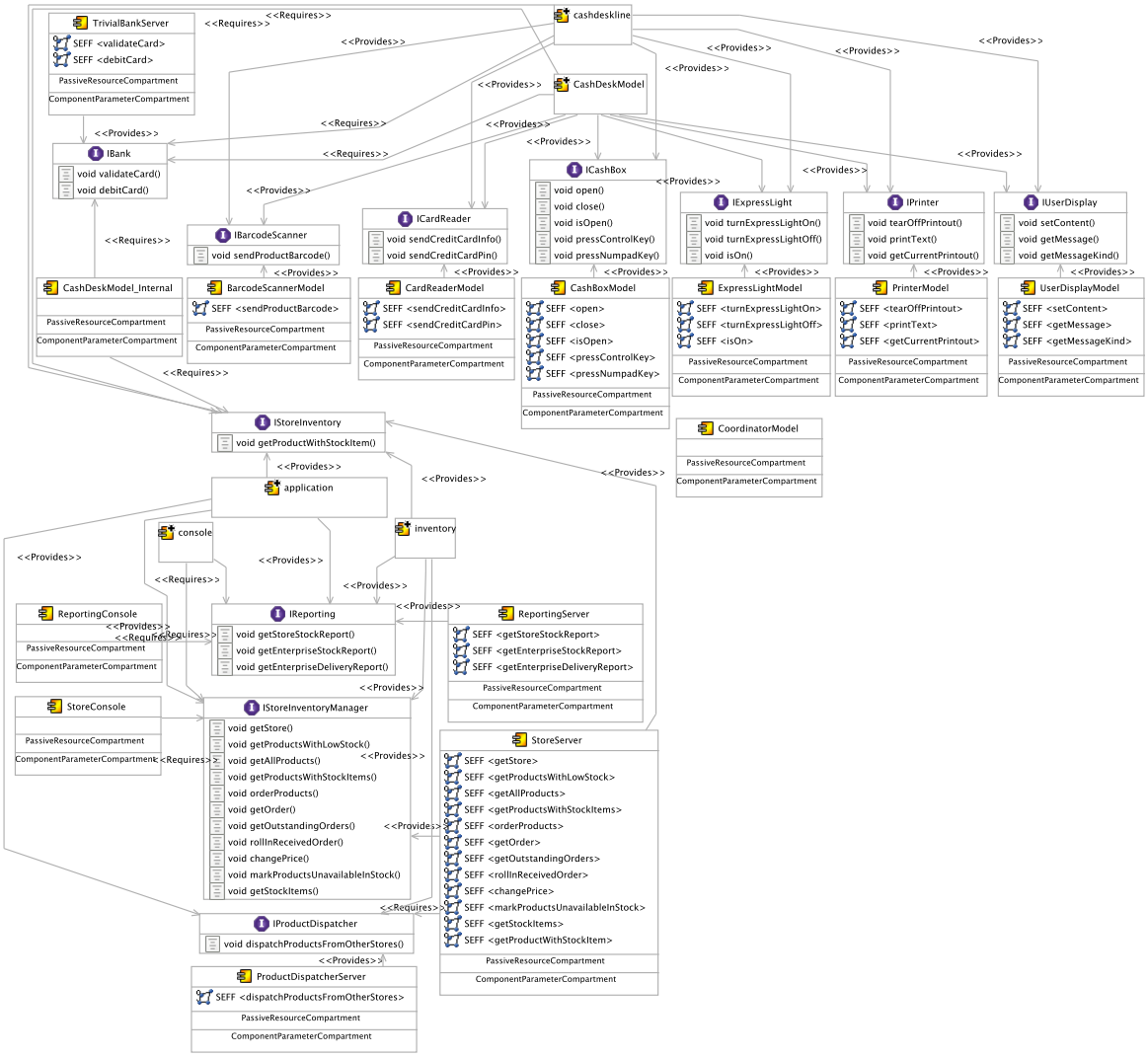


Figure 10.13: PCM Repository diagram of the CoCoME system, as it is extracted in the case study. The diagram's layout has been set manually.

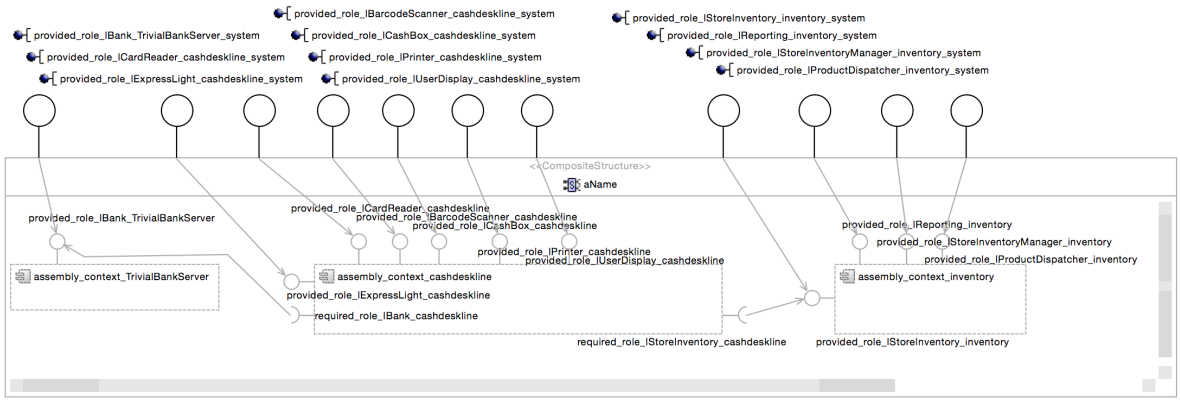


Figure 10.14: PCM System diagram of the CoCoME system, as it is extracted in the case study. The diagram's layout has been set manually.

study in Section 10.2. Therefore the same meta model, notations, and architecture model transformations between the architecture implementation language and the IAL are applied.

Translations between the IAL and UML already exist from the JACK case study (see Section 10.1). These transformations are, however, project-specific in their handling of composite components. Therefore a new, more general TGG was derived from the JACK case study to translate between the IAL meta model and the UML. The TGG in this case study translate the architecture, basic and composite components, provided and required interfaces, and their operations. Events have not been translated in this case study. For also translating events, it would be necessary to create more TGG rules. Figure 10.15 shows the result of the translation: a composite structure diagram which shows the topmost components: *TrivialBankServer*, the composite components *cashdeskline* and *inventory*, and their provided and required interfaces and subcomponents. The arrows with the keyword *use* are *Usage* relations in UML. Codeling only extracts the model information. The layout has been applied manually. The arrows without a keyword are *ComponentRealization* relations in UML. The complete TGG comprises ten rules. They can be found on the data medium associated with this thesis (see Appendix B).

This case study shows that the change of an architecture specification language is possible with CoCoME, and how this can be achieved. For changing the targeted specification language in comparison to the CoCoME case study in Section 10.2, the only changes necessary are new translations between the IAL and the targeted specification language. These translations can be reused for other projects.

10.4 Case Study: Implementation Language Migration

When an architecture implementation language evolves, or a new architecture implementation language emerges, an existing software system can be migrated to another architecture implementation language using Codeling. In this case study, an IAL view upon the CoCoME system (see Section 10.2.1) is extracted and an architecture implementation in JEE is generated. This is a case of architecture implementation language migration.

In this case study, only the steps 1 (Program Code to Translation Model), 2 (Inter-Profile

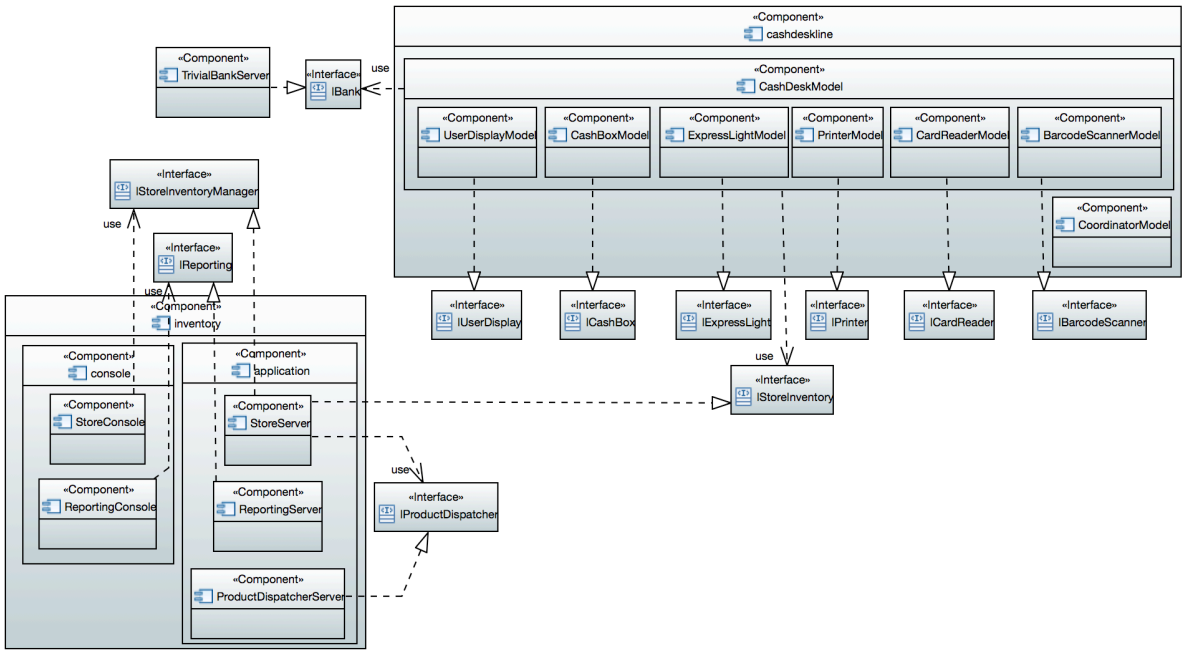


Figure 10.15: The CoCoME architecture in an UML composite structure diagram

Transformations), and 6 (Translation Model to Program Code) are executed. Step 1 of this case study is equal to step 1 of the CoCoME case study in Section 10.2. In step 2 instead of targeting the UML for translation, the architecture implementation language JEE is chosen. JEE provides only a flat component hierarchy, while the CoCoME architecture implementation language uses a scoped component hierarchy. Therefore other inter-profile transformations are executed. Steps 3 to 5 are omitted in this case study, because no architecture specification language is involved. In step 6 the same TGG is applied as in the JACK case study in Section 10.1. This includes the same architecture implementation language meta model and notations.

The translation results in a new project within the Eclipse IDE, with an architecture skeleton of CoCoME in JEE 7. The project contains request scoped CDI beans for each component, following the translation for new components as shown in Figure 10.6. The beans provide their respective interfaces and implement the necessary operations. Required interfaces are implemented via CDI's annotation `Inject`. The operation bodies are empty, because the operation content is not part of the translation between the architecture implementation languages and the IAL. All types are located inside a package named `new_architecture_elements`. The CoCoME meta model (see Section 10.2.2) does not consider namespaces. Therefore no namespaces can be propagated to the IAL. Namespaces could be added by adding notations for the respective IAL elements.

The parent-child relationship between components in the CoCoME architecture cannot be implemented in JEE, because JEE uses flat component hierarchies. In step 6, therefore a new notation has been added to the notations shown in Table 10.2, to integrate the parent-child relationship between components. Listing 10.8 shows the implementation of the composite component *CashDeskModel*. It implements all provided interfaces of its children, and delegates

the calls to their respective implementations. All child components are referenced using the CDI annotation `Inject`. The annotation `Child` has been defined in this case study to be the notation for representing the parent-child relationship in JEE program code, following the Annotated Member Reference mechanism. The original CoCoME implementation, the migrated CoCoME implementation, and all required transformations can be found on the data medium associated with this thesis (see Appendix B).

```
package new_architecture_elements;                                                    1
                                                                                      2
import javax.enterprise.context.RequestScoped;                                        3
                                                                                      4
@RequestScoped                                                                        5
public class CashDeskModel implements CashBox, ExpressLight, BarcodeScanner, Printer, 6
    UserDisplay, CardReader {
  @Inject @Child CashBoxModel cashBoxModel;                                           7
  @Inject @Child ExpressLightModel expressLightModel;                                 8
  @Inject @Child BarcodeScannerModel barcodeScannerModel;                             9
  @Inject @Child PrinterModel printerModel;                                           10
  @Inject @Child UserDisplayModel userDisplayModel;                                   11
  @Inject @Child CardReaderModel cardReaderModel;                                     12
                                                                                      13
  public void open(){ cashBoxModel.open(); };                                         14
  public void close(){ cashBoxModel.close(); };                                       15
  public boolean isOpen(){ return cashBoxModel.isOpen(); };                           16
  [...]                                                                               17
}                                                                                     18
```

Listing 10.8: Excerpt of the *CashDeskModel* as request scoped CDI bean. The code has been reformatted for readability reasons.

This case study shows that architecture implementation migration is possible with CoCoME. The implementation migration does not transfer the contents of the entry points to the new implementation. These contents need to be manually transferred, which decreases the use of the tool in this migration scenario. For changing the implementation language, the respective architecture model transformations and translations from the Model Integration Concept need to exist or be created. For this case study the same translations are used as in the case studies in Section 10.1 (the JEE translations) and Section 10.2 (the CoCoME translations).

10.5 The Bidirectionality of Transformations

The Explicitly Integrated Architecture Process requires the existence of transformations between program code, architecture implementation models, and architecture specification models within the Model Integration Concept and the architecture model transformations. These transformations should be designed to be bidirectional. Bidirectional transformations will reliably recreate the source model when transforming from a source model to a target model and back again. The implementation of these transformations for the Explicitly Integrated Architecture Process are specific to architecture languages. When the transformations are not designed bidirectionally, a roundtrip from the code to an architecture language and back without changing the target model might result in semantically changed code.

To support the implementation of bidirectional transformations in both the Model Integration Concept and the architecture model transformations, the implemented tools provide

libraries and frameworks. The implementation of the Model Integration Concept includes a framework for bidirectional model-code transformations (see Section 9.3.4). The framework includes transformation types that aggregate both directions within one Java type. This structure suggests the development of bidirectional transformations. Based upon that framework, bidirectional generic transformations for notations are made available in a library. When these transformations use predefined abstract transformations for integration mechanisms, they are automatically implemented bidirectionally.

For the implementation of architecture model transformations, Java helper types exist to execute transformations using HenshinTGG. While triple graph grammars are not necessarily bidirectional, they are meaningful candidates for bidirectional model-to-model transformations. The use of bidirectional transformations between the code and architecture specification models is therefore not enforced but assumed and supported. Therefore the requirement R4 can be seen as fulfilled. The transformations implemented throughout this thesis use these libraries and framework for creating bidirectional transformations. Not every transformation has been developed bidirectionally. The original code of CoCoME inconsistently realizes architectural concepts. This makes bidirectional transformations hard to achieve, However, bidirectionality is not necessary in the CoCoME-based case studies of this thesis.

10.6 Resource Demand

The execution of the case studies suggested that the transformations can require considerable execution times. The translation from the code to a UML representation in the JACK case study required about 350 seconds on a MacBook Pro, with a 2.9 GHz Intel Core i7 CPU and 8 GB 1600MHz DDR3 memory. The backwards translation of the changed model to changes in the code required about 380 seconds. The code-to-model translations did not contain information about operation parameters, because the translation required multiple hours when parameters were also translated. The CoCoME case studies required 170 seconds for the translation to the PCM, and 140 seconds to the UML. The migration of CoCoME to JEE required about 210 seconds on the same machine. The tool Codeling uses a series of code-to-model, model-to-code, and model transformations, including triple graph grammars (TGGs), to achieve its goals. TGGs can be performance intensive. Forward and backward translation rules from TGGs have a polynomial space and time complexity $O(m \times n^k)$, where m is the number of rules, n is the size of the input graph, and k is the maximum number of nodes in a rule [SK08]. Therefore it is expected, that an increasing model size implies a higher resource demand.

A resource demand test was executed during the development of Codeling, to find the limitations of the implementation. The questions to be answered by the resource demand study are:

1. *Which architecture model sizes can be handled with a reasonable time and memory demand?*

2. *Which parts of the implementation require the most resources?*

The following sections first describe the setup of the resource demand study in Section 10.6.1. The results are shown and discussed in Section 10.6.2.

Project Size	1	10	20	30	40	50	60	70	80	90
Number of Model Objects	13	112	222	332	442	552	662	772	882	992
Project Size	100	200	300	400	500					
Number of Model Objects	1102	2202	3302	4402	5502					

Table 10.4: The number of model object in dependency to the notated project size of the architecture implementation in the resource demand study

10.6.1 Study Setup

The study is set up as follows: A simple architecture implementation has been prepared. The architecture implementation is based on the EJB architecture implementation language, that has also been used in the JACK case study (see Section 10.1). The program code of the implementation is shown in Listings 10.9 and 10.10. The architecture implementation comprises one project with two beans in the same package. The stateless EJB Session Bean `package1.Provider1` contains two empty operations `operation1a` and `operation1b`. The stateless EJB Session Bean `Requirer1` also contains two operations `operation1a` and `operation1b`. Additionally, it contains a reference to the provider using the `EJB` annotation. This project contains 13 architecture elements in the architecture implementation language model: 1 *Architecture*, 1 *Archive*, 1 *Namespace*, 2 *Session Beans*, 4 *Operations*, and 4 *Operation Parameters* as return types.

```
package package1;                          1
                                           2
import javax.ejb.Stateless;                3
                                           4
@Stateless                                 5
public class Provider1 {                   6
  public void operation1a() {}             7
  public void operation1b() {}             8
}                                          9
```

Listing 10.9: The `Provider` type in the resource demand study

```
package package1;                          1
                                           2
import javax.ejb.EJB;                      3
import javax.ejb.Stateless;                4
                                           5
@Stateless                                 6
public class Requirer1 {                   7
                                           8
  @EJB                                     9
  Provider1 provider;                      10
  public void operation1a() {}             11
  public void operation1b() {}             12
}                                          13
```

Listing 10.10: The `Requirer` type in the resource demand study

To evaluate the memory and time demand for increasing input model sizes, increasingly bigger projects were generated based on that *basic project*. THe basic project has 1 provider-requirer pair, hence the project size is declared to be "1". The project size was increased as follows: First the basic project is copied to a new directory with a name that includes the targeted project size. Second, the project meta data was updated to reflect the new name. This is required for the projects to be usable within the Eclipse IDE. Then the package *package1* was copied n times, where n is the targeted project size. The name of the newly created packages is `packagei` with $i = 1..n$. The numbers in the name and file name of the `Provider` and `Requirer` types were changed accordingly. Table 10.4 shows the number of architecture elements in the input model for each generated project size. The project size n therefore means that the project contains: 1 *Architecture*, 1 *Archive*, n *Namespaces*, $2n$ *Session Beans*, $4n$ *Operations*, and $4n$ *Operation Parameters* as return types. The number of elements in a project of the size n is $11n + 2$.

Independent variables in the study are the amount of elements in the architecture implementation. Dependent variables are the translation time and the memory demand in terms of the Java heap size, that is used during the translation. Each project was translated with Codeling from code via the JEE meta model (see Section 10.1.2) to a UML representation in an automated JUnit[4] test case. The transformations of the JACK case study (see Section 10.1) were used. Each project size was translated five times. The automated tests measured the execution time, and stored the result and intermediate models. A test run included the automated execution of five translations in a sequence of Each test run was executed manually, and profiled with JProfiler[5]. JProfiler was configured to only collect a minimal data set (including the CPU load and heap size), resulting in a small footprint during the measurement. Before each measurement, a warm-up phase was executed, in which the basic project was translated once. This ensures that all meta models and translation rules are loaded before the measurement. After the measurement, the heap size histogram was exported as CVS file. The histogram relates the used heap size to the uptime of the Java process under test. For finding the maximum heap size of the translation in the histogram, the automated tests also stored the uptime of the Java process at the start of the translation, and the uptime at the end of the translation. Therefore the size of the used heap can be ignored for the parts of the test case, when no translation was executed.

The computer used for the measurements was a MacBook Pro, with a 2.9 GHz Intel Core i7 CPU and 8 GB 1600MHz DDR3 memory. For ensuring that the Java runtime has enough heap available, the following arguments were provided to the Java Virtual Machine: `-Xmx4g -Xms4g`. The virtual machine thus always had 4 GB of memory available for the heap.

10.6.2 Test Results and Discussion

The results of the measurements are aggregated in the Figures 10.16 for the duration of the translation, and 10.17 for the maximum heap size during the translation. Both show boxplot [HDFt17] diagrams, that group the results by the project size on the x-axis. For each value in the x-axis, the box extends from the lower to the upper quartile. The line within a box shows the median value. The vertical lines, that extend the boxes (the *whiskers*) show the extension of the lower and the upper quartile. Outliers are shown as circles below or above the whiskers. The single data points of all measurement runs can be found on the data medium attached to this thesis (see Appendix B). The x-axis of these diagrams is non-linear.

The resource demand study has shown that memory demand is in general not a problem in Codeling. Even though the Java Virtual Machine had 4 GB of memory available for the heap, the memory usage did not increase significantly between the increasing project sizes. The translation used a maximum of 2 GB of memory. For this project size the time required for the translation is more critical than the memory demand. During each single test run, the heap required for the translation increased slowly, suggesting that a memory leak might exist. In translations with greater input size, the memory demand in some test runs was decreased. The test runs with the project size 400 and 500 had a lower median memory demand, than the test run with the size 300, while also having a wider spread. We believe that during some of

[4] JUnit – A framework for automated tests in Java – `http://junit.org`

[5] JProfiler – A profiling tool for Java programs – `https://www.ej-technologies.com/products/jprofiler/overview.html`

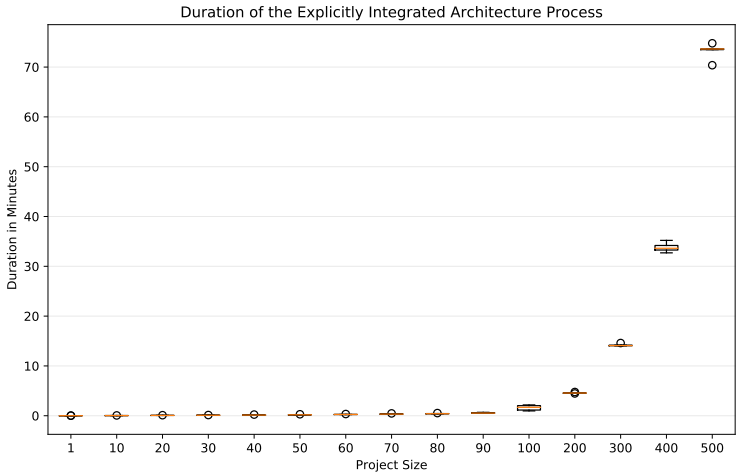


Figure 10.16: Boxplot diagram of the time required for the translation (with non-linear x-axis)

the long running test runs the garbage collection of the Java virtual machine cleaned the heap. It has to be stated that the memory footprint includes the complete Eclipse platform, that is used to execute the translations.

The bottleneck of the implementation is the time resource demand of the TGGs. With an increasing model size, the time required for executing the TGGs increased exponentially (as expected). Figure 10.18 shows the CPU load during the translation of a project with the size 300. The figure highlights the steps of the Explicitly Integrated Architecture Process in the time line. The CPU load shows that the most time is spent in the TGG transformations, and that these transformations create a load of about 25% on the CPU. The computer used for the case study has 4 CPU cores. It can be assumed that these transformations are executed in a single thread.

The questions, on which the resource demand test is based, can be answered as follows:

1. *Which architecture model sizes can be handled with a reasonable time and memory demand?*

   The memory demand of the implementation is considered to be not critical. The interpretation of the term "reasonable time" depends on the use case. For a daily use within an IDE by developers or architects, a translation time of some seconds, up to a small number of minutes can be acceptable, when the translation does not block other activities in the IDE. On the given computer, which can be seen as a typical development computer at the time of writing, the translation of an architecture implementation with 90 architecture elements took on average about 34 seconds. This can be considered to be within the desired range in this use case.

   When the implementation is used for documentation purposes, communication, or (possibly automated) analysis on a dedicated server, higher time resource demands may be acceptable. On the given computer, the translation of an architecture implementation with up to 200 elements took on average about 4,6 minutes. This can be considered to be within the a reasonable range in this use case.

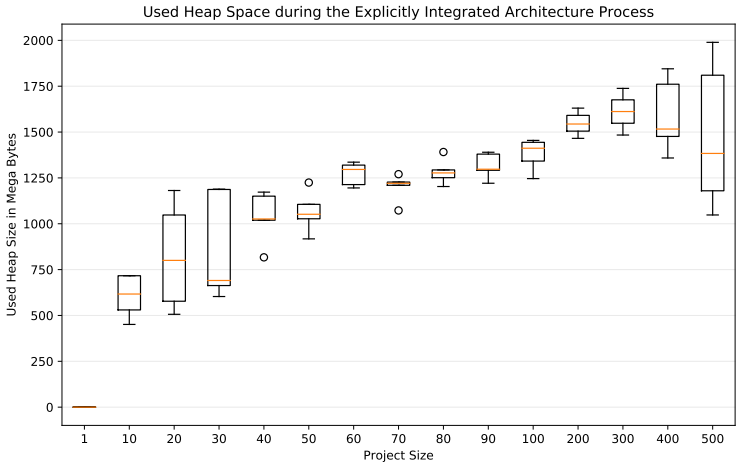


Figure 10.17: Boxplot diagram of the maximum heap required for the translation (with non-linear x-axis)

2. *Which parts of the implementation require the most resources?*

   The performance monitoring shows that the main time of the translations is spent within the TGG transformations. TGGs were chosen for Architecture Model Transformations in the Explicitly Integrated Architecture Approach, because they make it easy to create bidirectional transformations. The approach is not limited to TGGs, but can e.g. also use (non-TGG) graph transformation tools. Therefore it might be reasonable to compare the performance of the used HenshinTGG implementation with other TGG implementations, or to use another concept for bidirectional model transformations.

Independently from the implementation, TGGs have polynomial complexity with the size of the input graph as base and the maximum number of nodes in a rule as exponent. The input models in a Codeling translation are usually not easily changeable in a given project. We therefore encourage users that implement TGG rules for Codeling, to focus on developing rules with as few nodes as possible. All artefacts of the resource demand study—the basic project, the automated tests used for execution (which also generate the bigger projects as test input), all intermediate and result models, and all monitoring data sets, can be found on the data medium attached to this thesis (see Appendix B).

10.7 Discussion

The goal of the evaluation is to show whether the objective stated in Section 1.6 has been achieved. This is achieved by answering the corresponding questions.

R1 *Bridge the gap between software architecture specification languages and implementations thereof.*

For evaluating whether R1 is met, the following questions have to be answered:

Q1.1 Does a semantic equivalence relation exist for *explicit commonalities*?

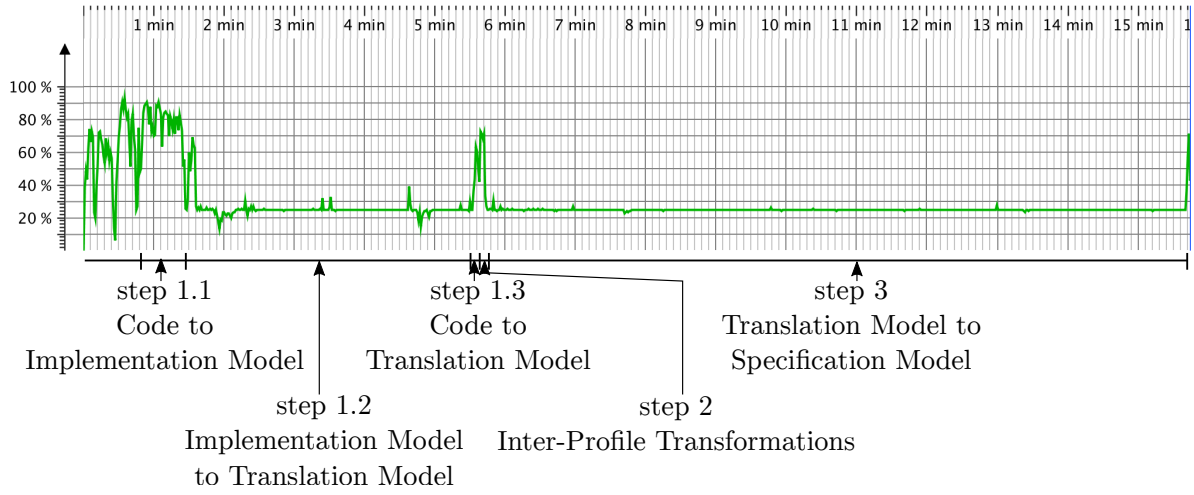


Figure 10.18: Histogram of the CPU load during the resource demand measurement of a project with the size 300

Explicit commonalities has been described in Section 1.4 to be first class elements in both the specification and the implementation. The Explicitly Integrated Architecture approach supports the definition of equivalence relations for explicit commonalities with these means.

The existence of semantic equivalence relations for explicit commonalities can be argued in the context of specific language integrations of the Explicitly Integrated Architecture approach. Explicit commonalities have been translated in each case study. An example of such a translation is the translation of session beans into UML components in the JACK case study (see Section 10.1). Both are considered declarations of component types in the context of this case study. An equivalence relation has been created by implementing transformation in the sense of the Model Integration Concept and architecture model transformations towards the IAL, as well as architecture model transformations between the IAL and the UML.

Q1.2 Does a semantic equivalence relation exist for *specifications that are translated into composed implementation structures*?

These commonalities between architecture implementation and specification languages have been described in Section 1.4 first class elements in a specification, that can be mapped to composed structures in the architecture implementation, which are considered equivalent. The Explicitly Integrated Architecture approach supports the definition of equivalence relations for the said commonalities.

Analogously to *Q1.1*, the existence of semantic equivalence can be argued in the context of specific language integrations of the Explicitly Integrated Architecture approach. Such commonalities have been translated in each case study. An example of such a translation is the translation of (childless) components in the CoCoME case study (see Section 10.2) into a basic component of the PCM. Both are considered declarations of component types in the context of this case study, but the implementation of such a component in CoCoME is a complex of composed program code elements. An equivalence relation has been defined using transformation in the sense of the Model Integration Concept. The resulting model element is translated to the PCM via the IAL using architecture model transformations.

R2 *Take the differences of architectural specification and implementation languages into account.*

Q2.1 Do program code representations exist for *specification details, that had no representation in the implementation* before?

Program code representations exist for specification details, that have no representation in implementation languages. In the Explicitly Integrated Architecture approach, translations between the IAL and the program code are explicitly considered.

In the JACK case study the UML has been used to annotate operations of interfaces with a time resource demand. The JEE meta model does not provide means to notate this quality attribute. The time resource demand represents a specification detail, that cannot be represented in the implementation language. The approach supports to execute bidirectional model-code transformations between the translation model and the program code. Such a translation was developed, based on an integration mechanism shown in Chapter 5, to create the mapping between the architecture specification model and the code. This translates a specification detail—the specified time resource demand—into a program code representation.

Q2.2 Are *implementation details, that have no representation in the specification*, preserved during changes in the specification?

Implementation details, that have no equivalent element or structure in the specification, are preserved during changes. The Explicitly Integrated Architecture approach handles such implementation details via the *entry points* in model notations (see Section 5.5). Entry points are places in the bidirectional model-code mappings, that can be enriched with arbitrary program code. Examples are method bodies in which a detailed behaviour can be implemented, or type declarations that can include arbitrary member attributes and operations.

In the JACK case study the component *MyAccount* is renamed in the UML diagram to *AccountDetailsView*. The change is propagated to the code, where it results in a renaming refactoring upon the type declaration that represents the UML component with an EJB session bean. The code in the type's body— which is part of the entry point— remains unchanged. The detailed implementation in the entry point survived the change operation within the specification. Therefor it is considered to be integrated with the specification.

R3 *Provide a single source of information for architecture descriptions.*

For evaluating whether R3 is met, the following question has to be answered:

Q3.1 Does a single source contain all implemented and specified architecture information?

The program code suffices as single source of information for architectural aspects. In the case studies, all desired architecture specification elements could be extracted from the program code. The Explicitly Integrated Architecture approach can be used to integrate architecture information that can be represented with languages that comply to the definitions of languages stated Chapter 5. The IAL is designed as an extensible language, so that arbitrary architectural concepts can be integrated, as long as they are expressible with the definition of languages and profiles (see Chapter 6).

R4 *Create bidirectional translations between the architecture specification and the implementation.*

Q4.1 Can specification views be derived from program code?

In the case studies JACK, CoCoME, and specification migration (see Section 10.3), specification views have been derived from the program code.

Q4.2 Are changes in the derived specification views propagated to the program code?

Changes in the derived specification view can be propagated to the program code. This bidirectionality is not enforced by the approach, but the use of TGGs and the implemented framework for transformations in the context of the Model Integration Concept support the bidirectionality. Bidirectionality has to be implemented in the specific language integrations. In the JACK case study, elements were added, changed, and removed the specification view. The program code is automatically changed by Codeling to reflect the changes in the specification. Section 10.5 discusses the implementation of bidirectionality in the approach.

R5 *Prepare for architecture specification and implementation language emergence and evolution.*

Q5.1 Can multiple architecture implementation and specification languages be used with the approach?

Multiple architecture implementation and specification languages can be used with the approach. In the different case studies, translations between multiple languages have been executed. The JACK case study translates between a subset of the Java Enterprise Edition and architectural concepts in the UML. The CoCoME case study translates between a project-specific architecture implementation language and the Palladio Component Model. The specification migration case study translates between CoCoME's architecture implementation language and the UML. Another modelling style was used in the UML in this case study, to better reflect the necessity of the specific project. In the implementation migration case study, the CoCoME implementation was translated into a the JEE architecture implementation language.

For adding further languages, the IAL is designed as intermediate language. For adding further languages, a meta model has to be created, and bidirectional architecture model transformations between the new language and the IAL have to be developed. For new architecture implementation languages, new bidirectional model-code transformations of the Model Integration Concept are necessary. Codeling supports the definition of these translations with libraries and a code generation tool. The IAL has been designed to be extensible with profiles, so that new architectural concepts can also be integrated into the translation.

Q5.2 Are languages weakly coupled with other languages in the approach?

Translations for languages, that are translated with the approach presented in this thesis, are weakly coupled via the IAL. The IAL serves as intermediate language, so that bidirectional transformations only have to be developed between the new language and the IAL. The translations developed for the case studies serve as demonstrators for this aspect.

As described above, *all questions for evaluating whether the requirements have been met can be answered with yes.* For some requirements, constraints have to be discussed: The evaluation shows that it is possible to bridge the gap between architecture specification languages and implementations thereof **(R1)**, by applying it to a set of case studies, which translate *explicit commonalities.*

The approach creates a single source of information for architecture descriptions **(R3)**. In the case studies this becomes visible, because the original information is in the program code, and the model information is derived from the code. In the second case study, information has to be added manually to the PCM for executing a simulation. This architecture information might also be integrated with code if desirable, by exploiting the integration mechanisms described in Section 5.6. Therefor notations have to be created for integrating IAL information with the code. Codeling is prepared for such operations, as it has been shown in the running example in Section 9.1 and Section 9.2.1, where component hierarchy information has been integrated with EJB code, although hierarchies are not part of that architecture implementation language.

The bidirectionality of translations **(R4)** depend on the implementation of the transformations. As argued in Section 10.5, bidirectionality is not enforced, but supported and encouraged by the approach. It has to be implemented in the specific language integrations. It has been shown in the case studies, that architecture specifications can be derived from the program code. The JACK case study specifically shows that changes in the specification can be propagated back to the program code. The case studies use subsets of the JEE, UML, and PCM, as well as a project-specific architecture implementation languages for translation. The UML and the JEE are complex languages, with great degrees of freedom how to use and combine their elements. In UML, even the semantics are in large parts subject to agreement within a team. In the case studies, specific subsets of these languages are used with a specific style of modelling. When not just subsets of the languages should be translated, but the complete languages, it can be expected, that bidirectionality is harder to achieve.

The approach has been evaluated in four case studies based on two different systems, one real world system in development, and the codebase of an academic case study, to show that the approach is prepared for multiple architecture specification and implementation languages. The handling of architecture specification and implementation language evolution and emergence **(R5)** is shown in two case studies in the sections 10.3 and 10.4. The translations of the language are loosely coupled via an extensible intermediate language.

At last, the approach should take the difference of architecture specification and implementation languages into account **(R2)**. This has been shown by executing the approach upon languages that model different aspects. Specification details without a representation in the program code could be integrated with the program code using the notations of the Model Integration Concept. The integration of implementation details with the specification has been shown with the JACK case study, where implementation details have not been lost during change operations.

The translation between the IAL and the UML in the first case study has shown that a generic transformation between the two languages has little value, but a project or at least domain-specific translation should be strived for. In the JACK case study, a project-specific translation was created to respect the wishes of the project team. The developed translations are a tailored set of translations of the third case study, which decreases the complexity of developing the translation. In the case study for architecture implementation language migration (see Section 10.4), a more generic translation has been developed for JEE. It seems to be promising to develop domain-specific architecture model translations, e.g. for business information systems in JEE, and adapt these blueprints to the project specific needs.

The evaluation has shown that the objective stated in Section 1.6 is achieved by the presented approach. In practice, four types of activities are enabled in this thesis:

Bottom-Up Extraction An architecture specification can be extracted from program code, as shown in the case studies in the sections 10.1, 10.2, and 10.3.

Top-Down Integration An architecture specification—or changes in an extracted specification—can be integrated with the underlying program code. The program code can be empty, so that a completely new implementation is created. The changing of an extracted specification has been shown in the JACK case study in Section 10.1.

Specification Migration When an architecture specification language evolves, or a new architecture specification language emerges, it might be desirable to use the new architecture specification language with an existing software system. This specification language migration process has been shown in the CoCoME to UML case study in Section 10.3.

Implementation Migration When an architecture implementation language evolves, or a new architecture implementation language emerges, it might be desirable to migrate an existing software system to the new architecture implementation language. This has been shown in the CoCoME to JEE case study in Section 10.4.

11 Conclusion

This chapter summarizes the contribution of this thesis in Section 11.1 and discusses the assumptions and limitations of the presented approach and its implementation in Section 11.2. Future work is discussed in Section 11.3.

11.1 Contributions

The essence of the challenge underlying this thesis is that software architecture information is spread across the implementation and adjacent model specifications, partly redundant, and without defined mapping between the implementation and the specification. This redundancy and lack of well-defined mapping is seen as a gap between the software architecture implementation and specification in this thesis. The objective of this thesis is stated in Section 1.6 as follows:

The development of concepts for bridging the gap
between software architecture specification and implementation

This thesis presents the following contributions for achieving this objective.

11.1.1 Model Integration Concept

This thesis presents the Model Integration Concept for integrating model information with program code. It defines *meta model notations* as formal mappings between meta model elements and program code structures, and *model notations* between model elements and program code structures. A set of integration mechanisms are described and discussed as templates for notations.

A tool has been implemented to generate program code based on the integration mechanisms. Based on a relation between meta model elements and integration mechanisms, it generates program code for meta model notations, transformations between model elements and program code structures, and runtime stubs. Program code for meta model notations is linked by program code for model notations to define that the code structures represent model elements. This generated code is reusable and can be made available in libraries. *Transformation types* are Java types of the transformation framework for the Model Integration Concept. The framework organizes the concurrent execution of transformations for the Model Integration Concept. The transformation types imperatively describe the translation between models and code structures. *Runtime stubs* for model notations can be used to create and manage runtime instances of models that have been expressed as program code structures. The model's runtime semantics can be implemented in these stubs.

The Model Integration Concept is the main driver to create a single source of architecture descriptions (requirement R3), and to create bidirectional translations between architecture specification and architecture implementation languages (R4).

11.1.2 Intermediate Architecture Description Language

The Intermediate Architecture Description Language (IAL) defined in this thesis is a language for a translation model between architecture implementation languages and architecture specification languages. It has a small kernel, which is extended by optional, and in parts mutually exclusive profiles. These profiles describe modelling aspects of architectures such as different kinds of component hierarchies, communication, or quality aspects. The language is designed to be extensible with further profiles, to express further aspects of software architecture modelling. This thesis provides translations between mutually exclusive profiles, which allows to translate translation models into multiple architecture implementation or specification languages, even if they use mutually exclusive kinds of architectural descriptions. No information is lost during such a translation.

11.1.3 Explicitly Integrated Architecture Process

The Explicitly Integrated Architecture Process describes an approach for creating a view upon program code, that is expressed with an architecture specification language. Changes in this view are be propagated to the code. The specification models can be derived from the program code, which renders the program code the leading source of information for the architecture.

The process makes use of the Model Integration Concept to extract architecture model information from the program code, and translates the findings into an architecture specification language via a translation model. Changes in the specification model are propagated via the translation model to the program code, which is changed accordingly.

The process uses the IAL as language for the translation model. As no information is lost during the translation between mutually exclusive profiles, the process can e.g. translate hierarchical component architectures into flat architectures without loosing the hierarchy information. The hierarchy still exists when model changes are propagated to the program code. Using the IAL and the Model Integration Concept, the Explicitly Integrated Architecture Process bridges the gap between architecture specification and implementation languages (R1). The IAL is able to express even mutually exclusive architecture information. The process can therefore take the differences of architectural specification and implementation languages into account (R2). It also enables the process to translate an implementation into different specification or implementation languages via its extensible set of profiles (R5).

A tool *Codeling* was implemented to execute the process. Codeling includes a framework for executing transformations of the Model Integration Concept and for adding architecture model transformations for architecture implementation and specification languages. In the evaluation, the process has been executed for two existing programs. One case study describes the translation of a real world program (JACK 3) that complies with the Java Enterprise Edition to UML composite structure diagrams. In the case study a new component is added to the program, which is then reflected in the program code. In the second case study, the original CoCoME implementation, a Java program following a project-specific architecture implementation style, is translated into a PCM model, which allows for executing performance simulations. In the third case study, an architecture specification language migration was executed, by translating the same CoCoME program code to UML as another specification language. The fourth case study shows an architecture implementation language migration, where the CoCoME program code is migrated to the JEE.

11.1.4 Bridging the Gap between Software Architecture Specifications and Implementations

Section 1.5 stated, which aspects of the software engineering process can benefit from a solution to the stated problem. Correspondingly, this thesis supports the software engineering process as follows:

The consistency of the architecture implementation and specification are improved: Codeling uses program code as single underlying model for architecture information. The consistency between the architecture implementation and specification are created by the construction and execution of architecture model transformations as well-defined, complete mappings. There is no need for consistency checks between the representations.

Consistency can not only be broken between multiple views, but also within a single view. The program code as single underlying model in this approach is a complex view. The Model Integration Concept supports the consistency of architecture information within the program code in two ways: first, the programming language's abstract syntax is used in combination with the validation features of compilers, and formally defined notations to enforce and support the consistency of dependencies between different artefacts. E.g. when the Model Integration Concept defines that a model object is represented as a type declaration, and its reference is represented with a member of that type, it ensures, that the ownership is unambiguously defined. Second, the proximity of information is used to support consistency, e.g. when performance annotations are attached to operations, it suggests to revisit the performance declaration, when the operation's content changes. Codeling therefore enforces and supports the consistency of architecture views by using the approach of a single underlying model.

Architecture implementations and specifications survive language evolution: When architecture implementation or specification languages evolve, or new languages emerge, a migration might become necessary. Codeling supports the migration between architecture languages with well-defined mappings between the implemented or specified architecture and the IAL in terms of model transformations. New or evolved languages can be included in the Explicitly Integrated Architecture Process by defining translations between the corresponding language and the IAL. When necessary, new IAL profiles can be created. Therefore Codeling makes the handling of architecture meta model evolution easier with a structured process.

The understandability of the architecture is increased: In Codeling the program code is the single underlying model. Every other architecture view is derived from the program code. Therefore the program code is the only original source of architecture information. Translations into specification language are formally defined and can be executed for creating abstract architectural views upon the system. Codeling increases the understandability by reducing the number of views to be identified, found, and understood.

Four activities are possible using the approach presented in this thesis.

Bottom-Up Extraction Architecture specifications can be extracted from program code, despite the differences between architecture specification and architecture implementation languages.

Top-Down Integration Architecture specifications—or changes in an extracted specification—can be integrated with the underlying program code, also despite the differences between architecture specification and architecture implementation languages.

Specification Migration When an architecture specification language evolves, or a new architecture specification language emerges, the specification language used to represent the program can be changed.

Implementation Migration When an architecture implementation language evolves, or a new architecture implementation language emerges, the architecturally modelled aspects of the program can be migrated to the new architecture implementation language.

This thesis' objective—to create a concept for bridging the gap between architecture implementation and architecture specification languages—is considered to be achieved due to the successful evaluation of the approach.

11.2 Assumptions and Limitations

The following assumptions and limitations apply to the approach and its implementation.

Component-based Development

The programs developed, evolved, or maintained with this approach are assumed to be developed based on interconnected components. The Intermediate Architecture Description Language's kernel expects the definition of component types, which provide and require interfaces. Therefore the least common denominator of architectural information must describe these artefacts.

The approach does not require the involved languages to model components explicitly. It suffices when abstractions of the architecture implementation or specification language are mappable to components and interfaces. E.g. languages based on service-oriented architecture descriptions can be used when the services are mapped to components providing and optionally requiring interfaces.

It is also possible to not translate a complete architecture. For example the profile for state-machine-based component types can be extracted separately, to analyse and change the state machines in tools that only handle state machines, not architectures. Due to the properties of the IAL translation model, the information about the ownership of the state machine can be preserved.

Implementation with One Language

The Explicitly Integrated Architecture Process and its implementation assume that an underlying program's architecture is implemented using a single architecture implementation language.

In large programs this is often not true. In the current implementation of the tool, the implementation language for the complete architecture is declared by the user.

When multiple implementation languages are used, it must be necessary to distinguish between the architecture languages used in the program. Therefore unambiguous program code structures should be used to identify the language in use, e.g. based on IDE projects, subdirectories, or component type implementations. The mapping between program code and the IAL must then include translations for all of these languages. The user should have an interface to decide, which implementation language is to be used for each element in the translation. The approach could suggest a language, based on the code (when model views are created from the code) or based on the model element's context (when a changed model view is translated into code). In the end, the user must decide which language to use. This is necessary to change the implementation language of an existing element or to decide which implementation language should be used for new elements.

Definition of Programming Languages

The definition of programming languages is one of the foundations of the Model Integration Concept. It is inspired by current object-oriented programming languages, specifically Java. The integration mechanisms and notations used in this thesis make heavy use of annotations and interfaces in the definition of programming languages. Therefore the existing mechanisms cannot be mapped to languages which do not support these elements.

For such programming languages to be used with the approach, it is possible to adapt the definition of programming languages. New integration mechanisms and notations can be defined based a new definition of programming languages, without further implications upon the general approach.

Definition of Modelling Languages

The definition of modelling languages is based on a subset of Ecore. The definition implies a meta modelled modelling language. Therefore, the presented approach is not applicable for modelling languages, that are not based on meta models, but e.g. on ontologies, without creating a meta model first. In the case studies, that were executed during this thesis, meta models of the architecture implementation languages had to be created beforehand. In the JACK case study (see Section 10.1) the meta model was created based on the textual specification [Ora13b]. In the CoCoME case study (see Section 10.2) the meta model was created based on a manual program code analysis.

Four Case Studies

The evaluation has shown that the objective has been achieved, by executing four case studies. The JACK case study (see Section 10.1) is based on a real life program in development – independently from this thesis' author – at our academic working group. The CoCoME case studies are based on a common benchmark software for software architecture research. Both programs are from the domain of information systems, which implies that their architectures in general follow similar quality goals, and therefore differ from each other less than the architectures of programs of two different domains.

The approach should be evaluated with further real life programs of further domains, for getting more detailed insights regarding the applicability and the necessity to adapt the approach to industry needs.

Contents of Entry Points in the Implementation Migration

During the implementation migration scenario presented in the case study in Section 10.4, the contents within the entry points of model notations have not been transferred to the new implementation. Conceptually this content can be transferred by creating a mapping between the entry points of the architecture implementation languages. Technically, the tool Codeling should be extended to exploit such an explicit mapping, or to transfer entry point contents where possible. The mapping might not always be possible. E.g. when an architectural object is represented with a type declaration in the source language, and with an interface declaration in the target language, no feasible target exists to transfer member attributes or references to.

Performance Evaluation

The performance evaluation has been executed with one model, which has been extrapolated to simulate bigger models. The measured performance is therefore only valid for the given model with its specific characteristics. Such characteristics include the proportion between the number of objects and the number of references between objects, and the proportion between the fan-in and the fan-out of objects.

For gaining a deeper insight into the performance characteristics of the prototype implementation, the performance should be evaluated with further models, that have with different characteristics.

Project-Specific Translations

It is desirable to have generic transformations for specific architecture implementation and specification languages. The architecture model translations for architecture implementation languages developed during the case studies are generic in large parts. Some parts, e.g. the component hierarchy expressed in IDE projects in the JACK case study (see Section 10.1) are project-specific, due to the project-specific needs in that case study.

In general it seems to be feasible to maintain a generic translation for a generic language specification – for both implementation and specification languages – and adapt the transformation to project-specific needs. The case studies indicate that the more generic a language can be used, the more difficult it is to create generic translations. As the UML is flexible to use, hardly any generic transformations could be defined, but most of the translations are project-specific. This imposes the question, whether project-specific transformations must be part of the architectural description, and should therefore also be subject to the Explicitly Integrated Architecture Process. However, this does not seem feasible.

11.3 Future Work

Besides the resolution of the limitations as described above, the following describes possible future work based on the results of this thesis.

Examination of Validity Constraints when Interweaving Integration Mechanisms

Notations for representing model elements can be interwoven, when one program code element is part of multiple model notations. In the running example in Section 5.6.1, the state machine and the component type are both mapped to the Type Annotation mechanism. This allows for marking one single type at the same time a component type and its state machine, as it is done in the running example for the component type *CashDesk*. This has implications on the model notation. E.g. no references must exist, that target objects with equal names in the classes *ComponentType* and *StateMachine*, that are both translated using the Annotated Member Reference mechanism. In that case two member attributes would be created with the same name, which is not valid according to Constraint 10. When notations are interwoven, complex validity constraints arise. These constraints should be examined, for avoiding such conflicts during the application of the approach.

Integration of Further Architecture Languages

This thesis describes the approach for working with integrated architecture specifications, and evaluates the general approach with four case studies, which implies the development of transformations for multiple architecture implementation languages and architecture specification languages. The integration of further languages would enhance the utility of the implementation.

Therefor transformations for broadly used languages should be created based on their specifications. These transformations can serve as a basis for possibly project-specific derivations.

Integration of Patterns and Styles

Prescriptive architecture models often define constraints for the architecture. Some of these constraints are in the form of patterns and styles, that are used to build the architecture. Such patterns and styles imply certain behavioural and structural constraints upon the architecture. E.g. in a layered architecture [BMR+96], each component must belong to a layer, and the communication paths between the layers are constrained.

Patterns can be described as roles in architectures, as done e.g. in the work of Durdik [Dur16, Section 4.2.3.2]. In Durdik's thesis, component and connector roles can be assigned to component types and connectors in the architecture. With this basis, analyzers can validate, whether the patterns and styles are not violated. When the translation and analysis is executed within an IDE, it is also possible to inform developers about such violation during the program code development.

Consideration of Imperative Behaviour

In the current state of the approach and the corresponding tool, behaviour implementations within operation bodies are not considered during translations. The Model Integration Concept does not provide any abstract syntax elements for statements and expressions, and in the IAL neither the kernel, nor any existing profile contains elements for representing this kind of behaviour. For including behaviour implementations within operation bodies, the Model Integration Concept needs to be extended with corresponding elements and their relations. During the architecture model translations, the implemented behaviour could be translated

into a generic behaviour language representation and back. Such a language profile must be created for the IAL in that case.

From a practical point of view, the translation into a generic behaviour language seems to be not feasible. Instead, the implemented behaviour—including related member attributes and references—could be stored as text. The behaviour can then be analyzed, e.g. regarding its side effects on security relevant data, using language specific analysis tools; or directly translated into specification languages such as the abstract behaviour specifications SEFFs from the PCM. This approach would also increase the benefit of implementation language migrations, when the architecture implementation language is based on the same programming language, because the implemented behaviour could be transferred to the migrated architecture implementation. This would certainly not result in a reasonable functionality in most cases, because a relationship between the architecture implementation language and the implemented behaviour usually exists. It would, however, be a better starting point for migrating the behaviour, than the current state, where no behaviour implementation is transferred.

Software Engineering Process Integration

The implementation of Codeling allows for extracting a specification model on demand within the IDE. The extraction of model specifications could also be executed automatically in a continuous integration or continuous deployment process. This could make visualization and automated analysis of models available in a broader context of the software engineering process.

Run Time Support

The approach currently handles design time models. The generation of execution runtime stubs (see Section 9.3.5) is a basis for creating runtime fragments for model elements, which are related to an integration mechanism. This does not imply the automated creation of run time models.

The generated execution runtime stubs could be extended with functionality to maintain a run time model of the architecture, where each object, attribute, and reference has a relation to its type. The use of the implemented execution runtime fragments would then manage a run time model, which can be the basis e.g. for automated monitoring and adaptation.

Extension to Further Domains

The case studies presented in this thesis are both from the information systems domain. The architectures in this domain usually have specific common characteristics. This is a threat to the generalisability of the approach. In other application domains not only other implementation and specification languages are used. It is also expected to be necessary to declare further profiles in the Intermediate Architecture Description Language, that handle the architectural aspects modelled in these domains.

Further Realistic Case Studies

Of the four case studies presented in this thesis, the first (JACK 3, Section 10.1) has a real life program as a subject, while the others use an artificial, but close to realistic system as a basis. To raise the external validity, further case studies should be performed on real life programs.

Final Remarks

To conclude, this thesis bridges the gap between architecture implementation languages and architecture specification languages by creating formalized mappings between code structures, architecture implementation models, and architecture specification models. It builds upon component-based software engineering, meta modelling, and model transformations. With the presented approach, architectures can be extracted from the program code and represented in specification languages for analysis, simulation, and for understanding and changing the models on a high abstraction level. Changes can be automatically propagated to the program code, even if the architecture implementation language cannot express one some of the modelled aspects. The specification models are not necessary as separate artefacts anymore, which leaves the program code as leading source of architecture information.

This thesis can be seen as a step closer to tightly integrated views of different types in software engineering, where inconsistencies between elements are resolved automatically. It does point to limitations, where such a tight integration is not feasible, such as information originating from the run time of the software, but also shows the possibilities to integrate views, and that there is still room for a deeper integration. It is a vision of mine, that tightly integrated views, spanning from the expression of ideas to executed behaviour of specific structural instances, can be managed and related without unhelpful redundancy. When software systems can be designed, by using models of just the right abstraction, it will not be necessary to invent workarounds in programming languages for implementing such abstractions. Just as real life objects are often represented in object-oriented programming languages in professional software engineering today, I would like to see that the abstract structural, behavioural, and quality aspects of software can be developed in feasible languages, without the need to create additional abstractions for these aspects in today's programming languages. Current programming languages should be used for what they are good at: describing detailed, algorithmic behaviour. We can then focus on using appropriate languages for the more abstract aspects. This thesis contributes to this vision: We create consistent views upon architectural elements and their program code representations, with entry points for detailed design in program code. We can now develop architectural aspects with architecture languages and detailed design in programming languages.

Part IV

Appendix

A References

[AB14] Aakash Ahmad and Muhammad Ali Babar. A Framework for Architecture-driven Migration of Legacy Systems to Cloud-enabled Software. In Anna Liu, John Klein, and Antony Tang, editors, *Proceedings of the WICSA 2014 Companion Volume, Sydney, NSW, Australia, April 7-11, 2014*. ACM, 2014.

[ABJ+10a] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. *Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations*, pages 121–135. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[ABJ+10b] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2010.

[ACC+14] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. Extraction and evolution of architectural variability models in plugin-based systems. *Software and System Modeling*, 13(4):1367–1394, 2014.

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 187–197. ACM, 2002.

[AGCK14] Marwan Abi-Antoun, Andrew Giang, Sumukhi Chandrashekar, and Ebrahim Khalaj. The Eclipse Runtime Perspective for Object-Oriented Code Exploration and Program Comprehension. In Andrew P. Black, Jan S. Rellermeyer, and Tim Verbelen, editors, *Proceedings of the Workshop on Eclipse Technology eXchange, ETX 2014*, pages 3–8. ACM, 2014.

[Amb03] Scott W. Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. Wiley, 2003. `http://www.agiledata.org/essays/mappingObjects.html`.

[AP11] Josef Adersberger and Michael Philippsen. ReflexML: UML-Based Architecture-to-Code Traceability and Consistency Checking. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *Software Architecture - 5th European Conference, ECSA 2011, Essen, Germany, September 13-16, 2011. Proceedings*, volume 6903 of *Lecture Notes in Computer Science*, pages 344–359. Springer, 2011.

[AST+16] Zakarea Alshara, Abdelhak-Djamel Seriai, Chouki Tibermacine, Hinde-Lilia Bouziane, Christophe Dony, and Anas Shatnawi. Materializing Architecture Recovered from Object-Oriented Source Code in Component-Based Languages. In Bedir Tekinerdogan, Uwe Zdun, and Muhammad Ali Babar, editors, *Software Architecture - 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28 - December 2, 2016, Proceedings*, volume 9839 of *Lecture Notes in Computer Science*, pages 309–325, 2016.

[Aß03] Uwe Aßmann. Automatic Roundtrip Engineering. *Electronic Notes in Theoretical Computer Science*, 82(5):33–41, April 2003.

[BA96] Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*. Wiley-IEEE Computer Society Press, 1996.

[Bal11] Moritz Balz. *Embedding Model Specifications in Object-Oriented Program Code: A Bottom-up Approach for Model-based Software Development*. PhD thesis, Universität Duisburg-Essen, May 2011.

[BBC+10] Felix Bachmann, Len Bass, Paul Clements, David Garlan, James Ivers, M. Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, second edition, 2010.

[BBK03] Felix Bachmann, Len Bass, and Mark Klein. Deriving Architectural Tactics: A Step Toward Methodical Architectural Design. Technical Report CMU/SEI-2003-TR-004, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2003.

[BCJM10] Hugo Brunelière, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 173–174. ACM, 2010.

[BEE+10] Enrico Biermann, Hartmut Ehrig, Claudia Ermel, Ulrike Golas, and Gabriele Taentzer. *Parallel Independence of Amalgamated Graph Transformations Applied to Model Transformation*, pages 121–140. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[BET12] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal Foundation of Consistent EMF Model Transformations by Algebraic Graph Transformation. *Software & Systems Modeling*, 11(2):227–250, 2012.

[BG03] S. A. Bohner and D. Gracanin. Software Impact Analysis in a Virtual Environment. In *28th Annual NASA Goddard Software Engineering Workshop, 2003. Proceedings*, pages 143–151, December 2003.

[BGMO06] Steffen Becker, Lars Grunske, Raffaela Mirandola, and Sven Overhage. Performance Prediction of Component-Based Systems. In *Architecting Systems with Trustworthy Components*, pages 169–192. Springer, Berlin, Heidelberg, 2006. DOI: 10.1007/11786160_10.

[BHT+10] Steffen Becker, Michael Hauck, Mircea Trifu, Klaus Krogmann, and Jan Kofron. Reverse Engineering Component Models for Quality Predictions. In Rafael Capilla, Rudolf Ferenc, and Juan C. Dueñas, editors, *14th European Conference on Software Maintenance and Reengineering, CSMR 2010, 15-18 March 2010, Madrid, Spain*, pages 194–197. IEEE Computer Society, 2010.

[BKBR12] Barbora Buhnova, Heiko Koziolek, Franz Brosch, and Ralf Reussner. Architecture-Based Reliability Prediction with the Palladio Component Model. *IEEE Transactions on Software Engineering*, 38(6):1319–1339, 2012.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82:3–22, 2009.

[Blo08] Joshua Bloch. *Effective Java*. Addison Wesley, 2nd edition edition, May 2008.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - A System of Patterns*, volume 1. Wiley John + Sons, 1996.

[BO09] Dominik Birkmeier and Sven Overhage. On Component Identification Approaches – Classification, State of the Art, and Comparison. In *Component-Based Software Engineering*, pages 1–18. Springer, Berlin, Heidelberg, June 2009.

[Boh02a] S. A. Bohner. Software Change Impacts – An Evolving Perspective. In *Proceedings of the International Conference on Software Maintenance*, pages 263–272, 2002.

[Boh02b] Shawn A. Bohner. Extending Software Change Impact Analysis into COTS Components. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, Washington, DC, USA, 2002. IEEE Computer Society.

[Bow15] Judy Bowen. Creating Models of Interactive Systems with the Support of Lightweight Reverse-Engineering Tools. In Michael Nebeling, Jürgen Ziegler, and Laurence Nigay, editors, *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS 2015, Duisburg, Germany, June 23-26, 2015*, pages 110–119. ACM, 2015.

[BT12] Steffen Becker and Matthias Tichy. Towards Model-Driven Evolution of Performance Critical Business Information Systems to Cloud Computing Architectures. *Softwaretechnik- note = ISSN 0720-8928,*, 32(2), 2012.

[CKK08] Landry Chouambe, Benjamin Klatt, and Klaus Krogmann. Reverse Engineering Software-Models of Component-Based Systems. In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, CSMR '08, pages 93–102, Washington, DC, USA, 2008. IEEE Computer Society.

[COB14] Everton Cavalcante, Flávio Oquendo, and Thaís Vasconcelos Batista. Architecture-Based Code Generation: From π-ADL Architecture Descriptions

to Implementations in the Go Language. In Paris Avgeriou and Uwe Zdun, editors, *Software Architecture - 8th European Conference, ECSA 2014, Vienna, Austria, August 25-29, 2014. Proceedings*, volume 8627 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2014.

[Cor10] Diarmuid Corcoran. The Good, the Bad and the Ugly: Experiences with Model Driven Development in Large Scale Projects at Ericsson. In Thomas Kühne, Bran Selic, Marie-Pierre Gervais, and François Terrier, editors, *Modelling Foundations and Applications, 6th European Conference, ECMFA 2010, Paris, France, June 15-18, 2010. Proceedings*, volume 6138 of *Lecture Notes in Computer Science*, page 2. Springer, 2010.

[DGJ+16] Nondini Das, Suchita Ganesan, Leo Jweda, Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. Supporting the Model-Driven Development of Real-time Embedded Systems with Run-Time Monitoring and Animation via Highly Customizable Code Generation. In Benoit Baudry and Benoît Combemale, editors, *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016*, pages 36–43. ACM, 2016.

[DMG07] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Signature Series (Fowler). Addison-Wesley Professional, June 2007.

[DMWW15] Jim Davies, David Milward, Chen-Wei Wang, and James Welch. Formal model-driven engineering of critical information systems. *Science of Computer Programming*, 103:88–113, 2015.

[DP09] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, July 2009.

[DRMM+10] Davide Di Ruscio, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Alfonso Pierantonio. Developing Next Generation ADLs Through MDE Techniques. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 85–94, New York, NY, USA, 2010. ACM.

[Dur16] Zoya Durdik. *Architectural Design Decision Documentation through Reuse of Design Patterns*. PhD thesis, Karlsruher Institut für Technologie, 2016.

[DvdHT02] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In *Proceedings of the 24th International Conference on Software Engineering*, pages 266–276, New York, NY, USA, 2002. ACM.

[EB10] Jürgen Ebert and Daniel Bildhauer. Reverse Engineering Using Graph Queries. In Gregor Engels, Claus Lewerentz, Wilhelm Schäfer, Andy Schürr, and Bernhard Westfechtel, editors, *Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*,

volume 5765 of *Lecture Notes in Computer Science*, pages 335–362. Springer, 2010.

[EBM12] George Edwards, Yuriy Brun, and Nenad Medvidovic. Automated Analysis and Code Generation for Domain-Specific Models. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, WICSA/ECSA 2012, Helsinki, Finland, August 20-24, 2012*, pages 161–170. IEEE, 2012.

[EJB09] EJB 3.1 Expert Group. JSR 318: Enterprise JavaBeans™, Version 3.1. `http://jcp.org/en/jsr/detail?id=318`, December 2009.

[EJB13] EJB 3.2 Expert Group. JSR 345: Enterprise JavaBeans™, Version 3.2 EJB Core Contracts and Requirement. `http://jcp.org/en/jsr/detail?id=345`, April 2013.

[EKRW02] Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. GUPRO - Generic Understanding of Programs An Overview. *Electronic Notes in Theoretical Computer Science*, 72(2):47–56, November 2002.

[FHK+15] Stefan Feldmann, Sebastian J. I. Herzig, Konstantin Kernschmidt, Thomas Wolfenstetter, Daniel Kammerl, Ahsan Qamar, Udo Lindemann, Helmut Krcmar, Christiaan J. J. Paredis, and Birgit Vogel-Heuser. A Comparison of Inconsistency Management Approaches using a Mechatronic Manufacturing System Design Case Study. In *IEEE International Conference on Automation Science and Engineering, CASE 2015, Gothenburg, Sweden, August 24-28, 2015*, pages 158–165. IEEE, 2015.

[FS96] Anthony Finkelstein and Ian Sommerville. The Viewpoints FAQ. *Software Engineering Journal*, 11(1):2–4, January 1996.

[FWKVH16] S. Feldmann, M. Wimmer, K. Kernschmidt, and B. Vogel-Heuser. A Comprehensive Approach for Managing Inter-Model Inconsistencies in Automated Production Systems Engineering. In *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 1120–1127, August 2016.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GJM03] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering (second edition)*. Prentice Hall, 2003.

[GJS+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java Language Specification - Java SE 8 Edition. `http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`, February 2015.

[GMRS08] Vincenzo Grassi, Raffaela Mirandola, Enrico Randazzo, and Antonino Sabetta. KLAPER : An Intermediate Language for Model-Driven Predictive Analysis of Performance and Reliability. In *The Common Component Modeling Example*, volume 5153/2008, pages 327–356. Springer Berlin / Heidelberg, 2008.

[GMS05] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. From Design to Analysis Models: a Kernel Language for Performance and Reliability Analysis of Component-based Systems. In *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, pages 25–36, New York, NY, USA, 2005. ACM.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. ACME: An Architecture Description Interchange Language. In *Proc. of CASCON'97*, pages 169–183, 1997.

[GoP] The Go Programming Language Specification. `http://golang.org/ref/spec`. accessed 2017-12-16.

[GR11] Dominik Gessenharter and Martin Rauscher. Code Generation for UML 2 Activity Diagrams - Towards a Comprehensive Model-Driven Development Approach. In Robert B. France, Jochen Malte Küster, Behzad Bordbar, and Richard F. Paige, editors, *Modelling Foundations and Applications - 7th European Conference, ECMFA 2011, Birmingham, UK, June 6 - 9, 2011 Proceedings*, volume 6698 of *Lecture Notes in Computer Science*, pages 205–220. Springer, 2011.

[GRe01] Graph Repository Query Language (GReQL). `https://www.uni-koblenz-landau.de/en/campus-koblenz/fb4/ist/rgebert/research/Graphtechnology/graph-repository-query-language-greql`, 2001.

[Gro13] Object Management Group. Action Language for Foundational UML (Alf) – Concrete Syntax for a UML Action Language, Version 1.0.1. `http://www.omg.org/spec/ALF/1.0.1/`, September 2013.

[Gro14] Object Management Group. Model Driven Architecture (MDA) – MDA Guide rev. 2.0. `http://www.omg.org/cgi-bin/doc?ormsc/14-06-01`, June 2014.

[Gro15a] Object Management Group. OMG Systems Modeling Language (OMG SysML^{TM}), Version 1.4. `http://www.omg.org/spec/SysML/1.4/`, September 2015.

[Gro15b] Object Management Group. XML Metadata Interchange (XMI) Specification, Version 2.5.1. `http://www.omg.org/spec/XMI/2.5.1/`, June 2015.

[Gro16] Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML). `http://www.omg.org/spec/FUML/1.2.1/`, January 2016.

[HAM10] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting Program Comprehension with Source Code Summarization. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 223–226. ACM, 2010.

[HCM11] Maen Hammad, Michael L. Collard, and Jonathan I. Maletic. Automatically identifying changes that impact code-to-design traceability during evolution. *Software Quality Journal*, 19(1):35–64, March 2011.

[HDB10] M. O. Hassan, L. Deruelle, and H. Basson. A Knowledge-Based System for Change Impact Analysis on Software Architecture. In *2010 Fourth International Conference on Research Challenges in Information Science (RCIS)*, pages 545–556, May 2010.

[HDFt17] John Hunter, Darren Dale, Michael Firing, Eric Droettboom, and the Matplotlib development team. matplotlib.axes.Axes.boxplot - The Matplotlib API - Version 2.1.0, October 2017. `https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.boxplot.html`.

[HEGO10] Frank Hermann, Hartmut Ehrig, Ulrike Golas, and Fernando Orejas. Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars. In *Proceedings of the First International Workshop on Model-Driven Interoperability*, MDI '10, pages 22–31, New York, NY, USA, 2010. ACM.

[HKW+08] Sebastian Herold, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, and Christian Pfaller. CoCoME - The Common Component Modeling Example. In Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and František Plášil, editors, *CoCoME*, volume 5153 of *Lecture Notes in Computer Science*, chapter 3, pages 16–60. Springer-Verlag, 2008.

[HMMP10] Rich Hilliard, Ivano Malavolta, Henry Muccini, and Patrizio Pelliccione. Realizing Architecture Frameworks Through Megamodelling Techniques. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 305–308, New York, NY, USA, 2010. ACM.

[HNZ17] Thomas Haitzer, Elena Navarro, and Uwe Zdun. Reconciling software architecture and source code in support of software evolution. *Journal of Systems and Software*, 123:119–144, 2017.

[HRR16] Robert Heinrich, Kiana Rostami, and Ralf Reussner. The CoCoME Platform for Collaborative Empirical Research on Information System Evolution. techreport 2016,2, Karlsruhe Institute of Technology, Faculty of Informatics, 2016. ISSN 2190-4782.

[II05] Suhaimi Ibrahim and Norbik Bashah Idris. A Requirements Traceability to Support Change Impact Analysis. *Asean Journal of Information Technology, Pakistan*, page 345355, 2005.

[IIM06] Suhaimi Ibrahim, Norbik Idris, and Malcolm Munro. A Software Traceability Validation For Change Impact Analysis of Object Oriented Software. In *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers*, volume 1, pages 453–459, Las Vegas, Nevada, USA, June 2006.

[IIMD05] Suhaimi Ibrahim, Norbik Bashah Idris, Malcolm Munro, and Aziz Deraman. Integrating Software Traceability for Change Impact Analysis. *Int. Arab J. Inf. Technol.*, 2(4):301–308, 2005.

[ISO11] Systems and software engineering — Architecture description. ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000), 2011.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, jun 2008.

[JSR14] JSR 346 EG. Contexts and Dependency Injection for the Java EE platform. `http://jcp.org/en/jsr/detail?id=346`, April 2014.

[JSR17] JSR 365 EG. JSR 365 - Contexts and Dependency Injection for Java 2.0. `http://jcp.org/en/jsr/detail?id=365`, April 2017.

[KBB16] Dimitris Karagiannis, Robert Andrei Buchmann, and Dominik Bork. Managing Consistency in Multi-View Enterprise Models: an Approach based on Semantic Queries. In *24th European Conference on Information Systems, ECIS 2016, Istanbul, Turkey, June 12-15, 2016*, 2016.

[KEKT16] Christel Kapto, Ghizlane El-Boussaidi, Segla Kpodjedo, and Chouki Tibermacine. Inferring Architectural Evolution from Source Code Analysis - A Tool-Supported Approach for the Detection of Architectural Tactics. In Bedir Tekinerdogan, Uwe Zdun, and Muhammad Ali Babar, editors, *Software Architecture - 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28 - December 2, 2016, Proceedings*, volume 9839 of *Lecture Notes in Computer Science*, pages 149–165, 2016.

[KG12] Marco Konersmann and Michael Goedicke. A Conceptual Framework and Experimental Workbench for Architectures. In Maritta Heisel, editor, *Software Service and Application Engineering*, volume 7365 of *Lecture Notes in Computer Science*, pages 36–52. Springer Berlin Heidelberg, 2012.

[KG14] Marco Konersmann and Michael Goedicke. Integrating Protocol Contracts with Program Code - A Leightweight Approach for Applied Behaviour Models that Respect Their Execution Context. In *Behavior Modeling - Foundations and Applications, International Workshops, BM-FA 2009-2014, Revised Selected Papers*, pages 197–219, 2014.

[KH14] Imran Khan and Sajjad Haider. On building a consistent framework for executable systems architecture. *Journal of Systems and Software*, 98:155–171, 2014.

[KKG14] Marco Konersmann, Noyan Kurt, and Michael Goedicke. Integrating Protocol Contracts with Java Code. In *Proceedings of the 2014 Workshop on Behaviour Modelling-Foundations and Applications*, BM-FA '14, pages 3:1–3:10, New York, NY, USA, 2014. ACM.

[KKK10] Tae-hyung Kim, Kimun Kim, and Woomok Kim. An Interactive Change Impact Analysis Based on an Architectural Reflexion Model Approach. In *Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference*,

COMPSAC '10, pages 297–302, Washington, DC, USA, 2010. IEEE Computer Society.

[KMC12] Tomaz Kosar, Marjan Mernik, and Jeffrey C. Carver. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical Software Engineering*, 17(3):276–304, 2012.

[Kon14] Marco Konersmann. Rapidly Locating and Understanding Errors Using Runtime Monitoring of Architecture-carrying Code. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, RCoSE 2014, pages 20–25, New York, NY, USA, 2014. ACM.

[Kon16] Marco Konersmann. A Process for Explicitly Integrated Software Architecture. *Softwaretechnik-Trends*, 36(2), 2016. ISSN: 0720-8928.

[Kru95] P. B. Kruchten. The 4+1 View Model of architecture. *IEEE Software*, 12(6):42–50, November 1995.

[Lai13] Huu Loi Lai. Entwicklung einer Werkzeugumgebung zur Visualisierung und Analyse komplexer EMF- Modelltransformationssysteme in Henshin. Master's thesis, Tecnical University Berlin, May 2013.

[Lan17] Michael Langhammer. *Automated Coevolution of Source Code and Software Architecture Models*. Phd thesis, Karlsruhe Institute of Technology, February 2017.

[Leh11a] Steffen Lehnert. A Review of Software Change Impact Analysis. Technical report, Technische Universität Ilmenau, 2011. urn:nbn:de:gbv:ilm1-2011200618.

[Leh11b] Steffen Lehnert. A Taxonomy for Software Change Impact Analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 41–50, New York, NY, USA, 2011. ACM.

[LK15] Michael Langhammer and Klaus Krogmann. A Co-evolution Approach for Source Code and Component-based Architecture Models. *Softwaretechnik-Trends*, 35(2), 2015. ISSN 0720-8928.

[LMT09] Francisco J. Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of UML model consistency management. *Information and Software Technology*, 51(12):1631 – 1645, 2009.

[LMT+14] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. Assessing the State-of-Practice of Model-Based Engineering in the Embedded Systems Domain. In Jürgen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfrán, editors, *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, volume 8767 of *Lecture Notes in Computer Science*, pages 166–182. Springer, 2014.

[LMWK14] Philip Langer, Tanja Mayerhofer, Manuel Wimmer, and Gerti Kappel. On the usage of UML: initial results of analyzing open UML models. In Hans-Georg Fill, Dimitris Karagiannis, and Ulrich Reimer, editors, *Modellierung 2014, 19.-21. März 2014, Wien, Österreich*, volume 225 of *Lecture Notes in Informatics*, pages 289–304. GI, 2014.

[Loh17] Enno Lohmann. Explizite Integration und Durchsetzung von Architekturconstraints. mathesis, University of Duisburg-Essen, 2017. (in German).

[LRSS10] Chung-Horng Lung, Pragash Rajeswaran, Sathyanarayanan Sivadas, and Theleepan Sivabalasingam. Experience of building an architecture-based generator using GenVoca for distributed systems. *Science of Computer Programming*, 75(8):672–688, 2010.

[LWWC12] Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. EMF Profiles: A Lightweight Extension Approach for EMF Models. *Journal of Object Technology*, 11(1):8:1–29, April 2012.

[Mal10] Ivano Malavolta. A model-driven approach for managing software architectures with multiple evolving concerns. In Ian Gorton, Carlos E. Cuesta, and Muhammad Ali Babar, editors, *Software Architecture, 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Companion Volume*, ACM International Conference Proceeding Series, pages 4–8. ACM, 2010.

[Man03] Heiko Mantel. *A Uniform Framework for the Formal Specificationand Verification of Information Flow Security.* PhD thesis, Universität Des Saarlandes, 2003.

[MB02] Stephen J Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures.* Addison-Wesley Longman Publishing Co. Inc., 2002.

[MBG10] Marco Müller, Moritz Balz, and Michael, Goedicke. Representing Formal Component Models in OSGi. In Gregor Engels, Markus Luckey, and Wilhelm Schäfer, editors, *Software Engineering*, volume 159 of *LNI*, pages 45–56. GI, 2010.

[MBG11a] Marco Müller, Moritz Balz, and Michael Goedicke. Enriching Java Enterprise Interfaces with Formal Sequential Contracts. In *Proceedings of the Third Workshop on Behavioural Modelling*, BM-FA '11, pages 5–11, New York, NY, USA, 2011. ACM.

[MBG11b] Marco Müller, Moritz Balz, and Michael Goedicke. Enriching OSGi Service Interfaces with Formal Sequential Contracts. In Ralf H. Reussner, Alexander Pretschner, and Stefan Jähnichen, editors, *Software Engineering 2011 - Workshopband (inkl. Doktorandensymposium), Fachtagung des GI-Fachbereichs Softwaretechnik, 21.-25.02.2011, Karlsruhe*, volume 184 of *Lecture Notes in Informatics*, pages 41–46. GI, 2011.

[McD02] J. H. McDuffie. Using the architecture description language MetaH for designing and prototyping an embedded reconfigurable sliding mode flight controller. In *Proceedings. The 21st Digital Avionics Systems Conference*, volume 2, 2002.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In Wilhelm Schäfer and Pere Botella, editors, *Software Engineering — ESEC '95: 5th European Software Engineering Conference Sitges, Spain, September 25–28, 1995 Proceedings*, pages 137–153, Berlin, Heidelberg, 1995. Springer.

[MF10] Rebecca Parsons Martin Fowler. *Domain-Specific Languages*. The Addison-Wesley Signature Series (Fowler). Addison-Wesley Professional, September 2010.

[Mic] Microsoft Corporation. Windows Workflow Foundation. `http://msdn.microsoft.com/en-us/library/dd489441.aspx`. accessed 2017-12-16.

[MK16] Jens Holschbach Marco Konersmann. Automatic Synchronization of Allocation Models with Running Software. *Softwaretechnik-Trends, Proceedings of the 7th Symposium on Software Performance 2016 (SSP)*, 36(4):28–29, November 2016. ISSN: 0720-8928.

[MLM+13] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, 39(6):869–891, June 2013.

[MMPT10] Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Damien Tamburri. Providing Architectural Languages and Tools Interoperability Through Model Transformation Technologies. *IEEE Trans. Softw. Eng.*, 36(1):119–140, January 2010.

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In *IEEE Transactions on Software Engineering*, pages 18–28, 1995.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[MVG06] T.a Mens and P.b Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152(1-2):125–142, 2006.

[Mü10] Marco Müller. Applying Formal Component Specifications to Module Systems in Java. Master's thesis, Universität Duisburg-Essen, March 2010.

[NER01] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Making inconsistency respectable in software development. *Journal of Systems and Software*, 58(2):171–180, September 2001.

[NNWZ00] U. A. Nickel, J. Niere, J. P. Wadsack, and A. Zündorf. Roundtrip Engineering with FUJABA. In *Proc of 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany*, 2000.

[Obj08] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. `http://www.omg.org/spec/QVT/1.0/`, April 2008.

[Obj15] Object Management Group. OMG Unified Modeling Language TM (OMG UML), Version 2.5. http://www.omg.org/spec/UML/2.5, May 2015.

[Obj16] Object Management Group. OMG Meta Object Facility (MOF) Core Specification, Version 2.5.1. http://www.omg.org/spec/MOF/2.5.1, November 2016.

[OLB16] Flávio Oquendo, Jair C. Leite, and Thaís Batista. Executing Software Architecture Descriptions with SysADL. In Bedir Tekinerdogan, Uwe Zdun, and Muhammad Ali Babar, editors, *Software Architecture - 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28 - December 2, 2016, Proceedings*, volume 9839 of *Lecture Notes in Computer Science*, pages 129–137, 2016.

[Oqu04] Flavio Oquendo. π-ADL: An Architecture Description Language Based on the Higher-order Typed π-calculus for Specifying Dynamic and Mobile Software Architectures. *SIGSOFT Software Engineering Notes*, 29(3):1–14, May 2004.

[Ora13a] Oracle America, Inc. JavaServer Faces^{TM}Specification - Version 2.2. https://jcp.org/en/jsr/detail?id=344, April 2013.

[Ora13b] Oracle America, Inc. JavaTM Platform, Enterprise Edition (Java EE) Specification v7. https://www.jcp.org/en/jsr/detail?id=342, May 2013.

[Par72] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[PD10] Werner Putschögl and Bernhard Dorninger. Modelling Interactions for Automatic Execution Using UML Activity Diagrams. In Gregor Engels, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2010, 24.-26. März 2010, Klagenfurt, Österreich*, volume 161 of *Lecture Notes in Informatics*, pages 179–194. GI, 2010.

[PGS01] Doron A. Peled, David Gries, and Fred B. Schneider, editors. *Software Reliability Methods*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[Pie78] Robert A. Pierce. A Requirements Tracing Tool. In *Proceedings of the Software Quality Assurance Workshop on Functional and Performance Issues*, pages 53–60, New York, NY, USA, 1978. ACM.

[Pü16] Markus Püschel. Program Generation for Performance. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, page 1. ACM, 2016.

[RAK15] Muhammad Rashid, Muhammad Waseem Anwar, and Aamir M. Khan. Toward the tools selection in model based system engineering for embedded systems - A systematic literature review. *Journal of Systems and Software*, 106:150–163, 2015.

[RBH+16] Ralf H. Reussner, Steffen Becker, Jens Happe, Robert Heinrich, and Anne Koziolek. *Modeling and Simulating Software Architectures*. MIT Press Ltd, 2016.

[RCM14] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. Model-driven reverse engineering of legacy graphical user interfaces. *Autom. Softw. Eng.*, 21(2):147–186, 2014.

[RGB+14] Thomas Ruhroth, Stefan Gärtner, Jens Bürger, Jan Jürjens, and Kurt Schneider. Versioning and Evolution Requirements for Model-Based System Development. *Softwaretechnik-Trends*, 34(2), 2014. ISSN 0720-8928.

[RJ12] Thomas Ruhroth and Jan Jürjens. Supporting Security Assurance in the Context of Evolution: Modular Modeling and Analysis with UMLsec. In *IEEE: 14th International Symposium on High-Assurance Systems Engineering (HASE 2012)*. IEEE CS, October 2012.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[RRIP14] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Dealing with the Coupled Evolution of Metamodels and Model-to-text Transformations. In Alfonso Pierantonio, Bernhard Schätz, and Dalila Tamzalit, editors, *Proceedings of the Workshop on Models and Evolution co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014), Valencia, Spain, Sept 28, 2014*, volume 1331 of *CEUR Workshop Proceedings*, pages 22–31, 2014.

[RS13] Igor Rozanc and Bostjan Slivnik. Using Reverse Engineering to Construct the Platform Independent Model of a Web Application for Student Information Systems. *Computer Science and Information Systems*, 10(4):1557–1583, 2013.

[RSHR15] Kiana Rostami, Johannes Stammel, Robert Heinrich, and Ralf Reussner. Architecture-based Assessment and Planning of Change Requests. In *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*, QoSA '15, pages 21–30, New York, NY, USA, 2015. ACM.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[Sch94] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1994.

[Sch16] Jasper Schenkhuizen. Consistent Inconsistency Management: A Concern-Driven Approach. Master's thesis, Utrecht University, 2016.

[Sel98] Bran Selic. Using UML for Modeling Complex Real-Time Systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools*

for Embedded Systems, LCTES '98, pages 250–260, London, UK, UK, 1998. Springer-Verlag.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, volume 1. Prentice Hall Englewood Cliffs, 1996.

[SG14a] Mikus Wolter Susann Gottmann, Frank Hermann. Generation of Operational Rules. `https://github.com/de-tu-berlin-tfs/Henshin-Editor/wiki/Generation-of-Operational-Rules`, September 2014. accessed 2017-12-16.

[SG14b] Mikus Wolter Susann Gottmann, Frank Hermann. State Based Backward Propagation. `https://github.com/de-tu-berlin-tfs/Henshin-Editor/wiki/State%20Based%20Backward%20Propagation`, 2014. accessed 2017-12-16.

[SG14c] Mikus Wolter Susann Gottmann, Frank Hermann. State Based Forward Propagation. `https://github.com/de-tu-berlin-tfs/Henshin-Editor/wiki/State%20Based%20Forward%20Propagation`, 2014. accessed 2017-12-16.

[SGT10] Giuseppe Scanniello, Carmine Gravino, and Genny Tortora. Investigating the Role of UML in the Software Modeling and Maintenance - A Preliminary Industrial Survey. In *Proceedings of the 12th International Conference on Enterprise Information Systems (ICEIS'10)*, pages 141–148. SciTePress, 2010.

[SHC+11] Hui Song, Gang Huang, Franck Chauvel, Yingfei Xiong, Zhenjiang Hu, Yanchun Sun, and Hong Mei. Supporting runtime software architecture: A bidirectional-transformation-based approach. *Journal of Systems and Software*, 84(5):711–723, 2011.

[Sie16] Janet Siegmund. Program Comprehension: Past, Present, and Future. In *Leaders of Tomorrow Symposium: Future of Software Engineering, FOSE@SANER 2016, Osaka, Japan, March 14, 2016*, pages 13–20. IEEE Computer Society, 2016.

[SISV16] Tatiana E. Shulga, E. A. Ivanov, Maria D. Slastihina, and Nataliia S. Vagarina. Developing a Software System for Automata-Based Code Generation. *Programming and Computer Software*, 42(3):167–173, 2016.

[SK08] Andy Schürr and Felix Klar. 15 Years of Triple Graph Grammars. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Proceedings of the 4th International Conference on Graph Transformations*, volume 5214 of *Lecture Notes in Computer Science*, pages 411–425. Springer, September 2008.

[SLS18] Eugene Syriani, Lechanceux Luhunu, and Houari Sahraoui. Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures*, 52:43–62, 2018.

[SLY16] Madhusudan Srinivasan, Young Lee, and Jeong Yang. Enhancing Object-Oriented Programming Comprehension Using Optimized Sequence Diagram. In *29th IEEE International Conference on Software Engineering Education and*

Training, CSEET 2016, Dallas, TX, USA, April 5-6, 2016, pages 81–85. IEEE, 2016.

[SM16] Tamal Sen and Rajib Mall. Extracting finite state representation of Java programs. *Software and System Modeling*, 15(2):497–511, 2016.

[Smi07] Geoffrey Smith. *Malware Detection*, chapter Principles of Secure Information Flow Analysis, pages 291–307. Springer US, Boston, MA, 2007.

[SRHR15] Misha Strittmatter, Kiana Rostami, Robert Heinrich, and Ralf H. Reussner. A Modular Reference Structure for Component-based Architecture Description Languages. In Federico Ciccozzi, Patrizio Pelliccione, and Etienne Borde, editors, *Proceedings of the 2nd International Workshop on Model-Driven Engineering for Component-Based Software Systems co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2015), Ottawa, Canada, September 28, 2015*, volume 1463 of *CEUR Workshop Proceedings*, pages 36–41, 2015.

[ST07] Ali R. Sharafat and Ladan Tahvildari. A Probabilistic Approach to Predict Changes in Object-Oriented Software Systems. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, CSMR '07, pages 27–38, Washington, DC, USA, 2007. IEEE Computer Society.

[ST08] Ali R. Sharafat and Ladan Tahvildari. Change Prediction in Object-Oriented Software Systems: A Probabilistic Approach. *Journal of Software*, 3(5), May 2008.

[Str16] Michael Striewe. An architecture for modular grading and feedback generation for complex exercises. *Science of Computer Programming*, 129:35 – 47, 2016. Special issue on eLearning Software Architectures.

[SV12] Petr C. Smolik and Pavel Vitkovsky. Code Generation Nirvana. In Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitrios S. Kolovos, editors, *Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings*, volume 7349 of *Lecture Notes in Computer Science*, pages 319–327. Springer, 2012.

[SVC06] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006.

[SvdWJC16] Jasper Schenkhuizen, Jan Martijn E. M. van der Werf, Slinger Jansen, and Lambert Caljouw. Consistent Inconsistency Management: A Concern-Driven Approach. In Bedir Tekinerdogan, Uwe Zdun, and Muhammad Ali Babar, editors, *Software Architecture - 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28 - December 2, 2016, Proceedings*, volume 9839 of *Lecture Notes in Computer Science*, pages 201–209, 2016.

[SZ01] George Spanoudakis and Andrea Zisman. Inconsistency Management in Software Engineering: Survey and Open Research Issues. In *Handbook of Software Engineering and Knowledge Engineering*, pages 329–380. World Scientific Publishing Company, December 2001.

[The13] The Open Group. *ArchiMate*, 12 2013. Version 2.1.

[The14] The OSGi Alliance. OSGi Core. `https://osgi.org/download/r6/osgi.core-6.0.0.pdf`, June 2014.

[TMD09] R. N. Taylor, Nenad Medvidovic, and Irvine E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 1 edition, January 2009.

[vDPB14] Markus von Detten, Marie Christin Platenius, and Steffen Becker. Reengineering component-based software systems with Archimetrix. *Software and System Modeling*, 13(4):1239–1268, 2014.

[Voe10] Markus Voelter. Embedded Software Development with Projectional Language Workbenches. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part II*, volume 6395 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2010.

[vOvdLKM00] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, March 2000.

[WHR14] Jon Whittle, John Edward Hutchinson, and Mark Rouncefield. The State of Practice in Model-Driven Engineering. *IEEE Software*, 31(3):79–85, 2014.

[WMBK12] Rainer Weinreich, Cornelia Miesbauer, Georg Buchgeher, and Thomas Kriechbaum. Extracting and Facilitating Architecture in Service-Oriented Software Systems. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, WICSA/ECSA 2012, Helsinki, Finland, August 20-24, 2012*, pages 81–90. IEEE, 2012.

[WP10] Stefan Winkler and Jens von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software & Systems Modeling*, 9(4):529–565, September 2010.

[ZT13] Yongjie Zheng and Richard N. Taylor. A classification and rationalization of model-based software development. *Software and System Modeling*, 12(4):669–678, 2013.

B Data Medium Content

This thesis is accompanied with a data medium, which contains the artefacts developed in its context.

Dissertation This document as PDF document is contained in the main folder.

Program Code of the Developed Tools The program code of **Codeling** (see Section 9.2) is in the folder `sources/codeling`. At the time of publishing this thesis, it is also available online under `https://s3gitlab.paluno.uni-due.de/ADVERT/codeling`.

The program code of the **code generation tool** (see Section 9.3) is in the folder `sources/codeGenerator`. At the time of publishing this thesis, it is also available online under `https://s3gitlab.paluno.uni-due.de/ADVERT/CodelingLanguageIntegrator`.

Program Code, Transformations, and Models of the Running Example for Codeling The transformations, meta model libraries, execution runtime, the original program code, and the changed program code of the running example for Codeling in Section 9.2.1 are in the folder `sources/codeling/Language Integration/examples/running-example`. The extracted specification model, a changed specification model, and all intermediate models can be found in the folder `evaluation/running-example`.

Program Code of the Case Study Subjects The program code of **JACK 3**—in the state used in the case study in Section 10.1—is in the folder `evaluation/case-studies/jack3`. At the time of publishing this thesis, this program code is not publicly available online.

The program code of **CoCoME**, as it is used in the case study in Section 10.2, is in the folder `evaluation/case-studies/cocome`. The code has been adapted as described in Section 10.2. At the time of publishing this thesis, the original program code is available online under `https://github.com/cocome-community-case-study/cocome-plain-java`. The adapted program code is online under `https://s3gitlab.paluno.uni-due.de/ADVERT/case-study_cocome-plain-java`. The resulting program code of the implementation migration case study (see Section 10.4) can be found in the folder `evaluation/cocome-in-jee`.

Executables of the Developed Tools An Eclipse installation (for MacOS) and workspace for executing the case studies in Codeling and for executing the code generation tool on the running example are in the folder `executable`.

The workspace references the program code of the case study subjects for reevaluating the case studies. To do so, start the Eclipse installation using the workspace. The case studies can be run automatically, by executing the corresponding JUnit run configurations in Eclipse. They can also be run manually, by starting a client Eclipse with the run configuration `Codeling` in Eclipse. In the client Eclipse you can select the corresponding projects of the case studies (for JACK 3 the project `jack3-core`, `jack3-business`, and `jack3-webclient`, for CoCoME the project `cocome-impl`), and

open the context menu with a right click on the selection. Then use the menu item `Start Explicitly Integrated Architecture Process` to trigger a UI for selecting the corresponding languages for translation. The translation is started when the languages are chosen. When the translation is finished, the resulting model can be found in the file `architecture-carrying-code-temp/specification-model.xmi` within the workspace. When the model is changed as described in the case studies, and the changes are saved within Eclipse, Codeling automatically starts the translation towards the code.

For executing the code generation tool as described in the running example in Section 9.1, start a client Eclipse and right click the file `ejb-with-state-machine/metamodel/ejb-with-state-machine.ecore`. In the context menu select `Generate Explicitly Integrated Architecture Code`, load the integration mechanism mapping from the file `ejb-with-state-machine.example/model/Mechanisms-Mapping.xml` and click `Start` to generate the initial meta model notations, transformations, and runtime stubs. Please note that not all mechanisms have been implemented. Therefore not every mechanism can be translated. The generated program code with all necessary changes is already available in the top most ("host") Eclipse in the projects `ejb-with-state-machine.ial.mm` (the program code for meta model notations), `ejb-with-state-machine.transformation` (the transformations of the Model Integration Concept), and `ejb-with-state-machine.runtime` (the execution runtime stubs, extended with executional semantics).

The project `ejb-with-state-machine.example` uses the execution runtime in the Java type `org.codeling.example.ejbsm.CashDesk` (see Section 9.3.5). For executing the prepared execution runtime for EJBs with State Machines of the implementation's running example (see Section 9.1), execute the run configuration `EJB with State Machine Example` in the host Eclipse.

For executing the prepared model-code transformations, start a client Eclipse with the run configuration `Codeling`. Within this client Eclipse right click the project `ejb-with-state-machine.example` and use the context menu `Start Explicitly Integrated Architecture Process`. The resulting model can be found in the file `architecture-carrying-code-temp/specification-model.xmi` within the workspace. When the model is changed and the changes are saved within Eclipse, Codeling automatically starts the translation towards the code.

The translations for the JACK 3 case study (see Section 10.1) can be found in the following files in the client Eclipse:

- The meta model for JEE of the JACK 3 case study is in the file `org.codeling.lang.jee.metamodel/metamodel/jee7.ecore`. To see a graphical representation of the meta model, double click on the file `org.codeling.lang.jee.metamodel/metamodel/representations.aird`, expand the tree completely and double click the tree element *main*.
- The translations of the Model Integration Concept can be found in `org.codeling.lang.jee.transformation/src/main/java`.
- The TGG rules between the architecture implementation language and the Intermediate Architecture Description Language can be found in

`org.codeling.lang.jee.transformation/src/main/resources/AIL2IAL.henshin`. To see the TGG, open the file with the *TGG-Editor*.

- The TGG rules between the Intermediate Architecture Description Language and the UML can be found in `org.codeling.lang.jee.transformation/src/main/resources/AIL2IAL.henshin`. To see the TGG, open the file with the *TGG-Editor*.

The translations for the CoCoME case studies (see Sections 10.2, 10.3, and 10.4) can be found in the following files in the client Eclipse:

- The meta model for CoCoME is in the file `org.codeling.lang.cocome.metamodel/metamodel/cocome.ecore`. To see a graphical representation of the meta model, double click on the file `org.codeling.lang.cocome.metamodel/metamodel/representations.aird`, expand the tree completely and double click the tree element *main*.
- The translations of the Model Integration Concept can be found in `org.codeling.lang.jee.transformation/src/main/java`.
- The TGG rules between the architecture implementation language and the Intermediate Architecture Description Language can be found in the project `org.codeling.lang.cocome.transformation` in the file `/src/main/resources/AIL2IAL.henshin`. To see the TGG, open the file with the *TGG-Editor*.
- The TGG rules between the Intermediate Architecture Description Language and the PCM can be found in `org.codeling.lang.pcm/src/main/resources/PCM2IAL.henshin`. To see the TGG, open the file with the *TGG-Editor*.

Program Code of the Resource Demand Test Project and Test Results The program code of test project used in Section 10.6 can be found in the folder `sources/codeling/Examples/Resource_Demand/resource-demand-project_2`.
The data collected during the tests, all generated models, and the scripts for data aggregation and visualization can be found in the folder `evaluation/resource-demand`.

C Intermediate Language Profile Examples

C.1 Kernel

Figure C.1 shows an example model of the IAL kernel. Example 9 gives the formal definition of this model.

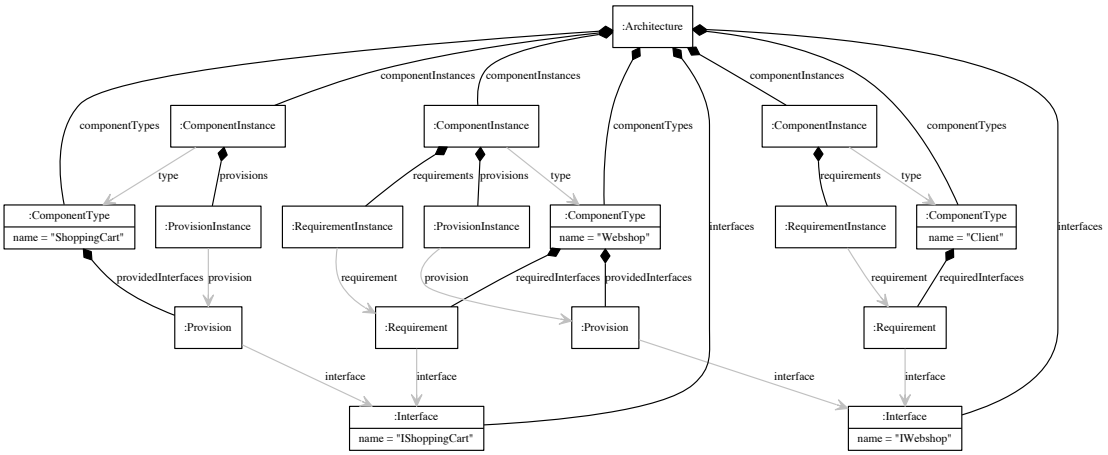


Figure C.1: Example model of the IAL meta model

Example 9: Example model of the IAL kernel

The exemplary model $M_{Kernel}^{Example}$, that instantiates the IAL kernel meta model, is formalized as follows. Figure C.1 accompanies the definition as an overview.

$$L_{Meta} = M_{Meta}^{Kernel}$$

$$\begin{aligned} O := \{ & o_{Architecture}, o_{IWebshop}, o_{IShoppingCart}, o_{ShoppingCart}, o_{Webshop}, o_{Client}, \\ & o_{ComponentInstance}^{ShoppingCart}, o_{ComponentInstance}^{Webshop}, o_{ComponentInstance}^{Client}, \\ & o_{Provision}^{ShoppingCart}, o_{Provision}^{Webshop}, o_{Requirement}^{Webshop}, o_{Requirement}^{Client}, \\ & o_{ProvisionInstance}^{ShoppingCart}, o_{ProvisionInstance}^{Webshop} o_{RequirementInstance}^{Webshop}, o_{RequirementInstance}^{Client} \} \end{aligned}$$

The elements are named as follows:

$name(o_{IWebshop})$ = IWebshop, $name(o_{IShoppingCart})$ = IShoppingCart,
$name(o_{Webshop})$ = Webshop, $name(o_{Client})$ = Client,
$name(o_{ShoppingCart})$ = ShoppingCart

The following values and targets are assigned to attributes and references:

$o_{Architecture}$.interfaces $\xrightarrow{references}$ IShoppingCart,
$o_{Architecture}$.interfaces $\xrightarrow{references}$ IWebshop,
$o_{Architecture}$.componentTypes $\xrightarrow{references}$ ShoppingCart,
$o_{Architecture}$.componentTypes $\xrightarrow{references}$ Webshop,
$o_{Architecture}$.componentTypes $\xrightarrow{references}$ Client,
$o_{Architecture}$.componentInstances $\xrightarrow{references}$ $o^{ShoppingCart}_{ComponentInstance}$,
$o_{Architecture}$.componentInstances $\xrightarrow{references}$ ComponentInstance^{Webshop},
$o_{Architecture}$.componentInstances $\xrightarrow{references}$ $o^{Client}_{ComponentInstance}$,
$o^{Webshop}_{ComponentInstance}$.type $\xrightarrow{references}$ Webshop,
$o^{Client}_{ComponentInstance}$.type $\xrightarrow{references}$ Client,
$o^{ShoppingCart}_{ComponentInstance}$.type $\xrightarrow{references}$ ShoppingCart,
ShoppingCart.providedInterfaces $\xrightarrow{references}$ $o^{ShoppingCart}_{Provision}$,
Webshop.providedInterfaces $\xrightarrow{references}$ $o^{Webshop}_{Provision}$,
Webshop.requiredInterfaces $\xrightarrow{references}$ $o^{Webshop}_{Requirement}$,
Client.requiredInterfaces $\xrightarrow{references}$ $o^{Client}_{Requirement}$,
$o^{ShoppingCart}_{Provision}$.interface $\xrightarrow{references}$ IShoppingCart,
$o^{Webshop}_{Provision}$.interface $\xrightarrow{references}$ IWebshop,
$o^{Webshop}_{Requirement}$.interface $\xrightarrow{references}$ IShoppingCart,
$o^{Client}_{Requirement}$.interface $\xrightarrow{references}$ IWebshop,
$o^{ShoppingCart}_{ComponentInstance}$.provisions $\xrightarrow{references}$ $o^{ShoppingCart}_{ProvisionInstance}$,
$o^{Webshop}_{ComponentInstance}$.provisions $\xrightarrow{references}$ $o^{Webshop}_{ProvisionInstance}$,
$o^{Webshop}_{ComponentInstance}$.requirements $\xrightarrow{references}$ $o^{Webshop}_{RequirementInstance}$,
$o^{Client}_{ComponentInstance}$.requirements $\xrightarrow{references}$ $o^{Client}_{RequirementInstance}$,

$$o^{ShoppingCart}_{ProvisionInstance}.\text{provision} \xrightarrow{references} o^{ShoppingCart}_{Provision},$$
$$o^{Webshop}_{ProvisionInstance}.\text{provision} \xrightarrow{references} o^{Webshop}_{Provision},$$
$$o^{Webshop}_{RequirementInstance}.\text{requirement} \xrightarrow{references} o^{Webshop}_{Requirement},$$
$$o^{Client}_{RequirementInstance}.\text{requirement} \xrightarrow{references} o^{Client}_{Requirement}$$

C.2 Operation Interfaces

Figure C.2 shows an example model of the profile *Operation Interfaces*. Example 10 gives the formal definition of this model.

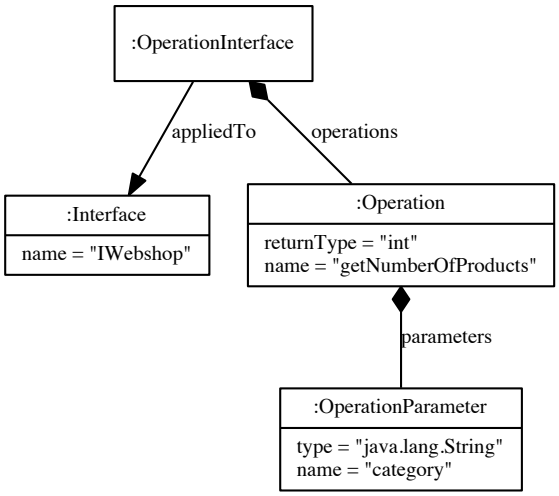


Figure C.2: Example application of the profile *Operation Interfaces*

Example 10: Example application of the profile *Operation Interfaces*

The exemplary profile application $M^{Example}_{OperationInterfaces}$, that instantiates the profile *Operation Interfaces*, is formalized as follows. Figure C.2 accompanies the definition as an overview.

$$P = P_{OperationInterfaces}$$

$$R = \{M^{Example}_{Kernel}\}$$

$$O := \{o_{getNumberOfProducts}, o_{category}\}$$

The elements are named as follows:

$$name(o_{getNumberOfProducts}) = \text{getNumberOfProducts},$$
$$name(o_{category}) = \text{category}$$

The stereotypes are applied as follows:

$$\text{OperationInterface} \xrightarrow{appliedTo} \text{IWebshop}$$

The following values and targets are assigned to attributes and references:

$$\text{getNumberOfProducts.returnType} \xrightarrow{hasValue} \text{int},$$
$$\text{category.type} \xrightarrow{hasValue} \text{java.lang.String},$$
$$\text{getNumberOfProducts.parameters} \xrightarrow{references} \text{category},$$
$$\text{IWebshop.operations} \xrightarrow{references} \text{getNumberOfProducts}$$

C.3 Event Interfaces

Figure C.3 shows an example model of the profile *Event Interfaces*. Example 11 gives the formal definition of this model.

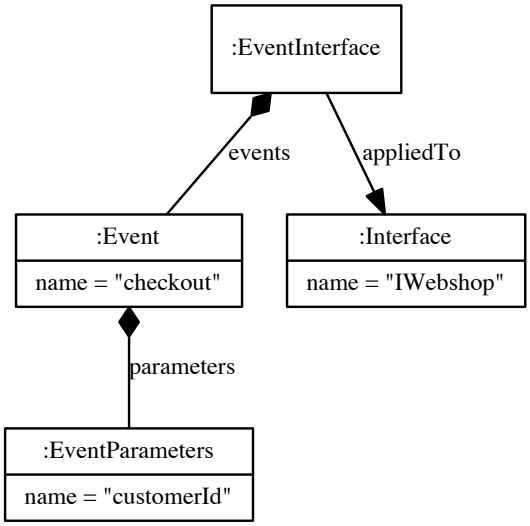


Figure C.3: Example application of the profile *Event Interfaces*

Example 11: Example application of the profile *Event Interfaces*

The exemplary profile application $M^{Example}_{EventInterfaces}$, that instantiates the profile *Event Interfaces*, is formalized as follows. Figure C.3 accompanies the definition as an overview.

$$P = P_{EventInterfaces}$$

$$R = \{M^{Example}_{Kernel}\}$$

$$O := \{o_{checkout}, o_{customerId}\}$$

The elements are named as follows:

$$name(o_{checkout}) = \text{checkout}, name(o_{customerId}) = \text{customerId}$$

The stereotypes are applied as follows:

$$\text{EventInterface} \xrightarrow{appliedTo} \text{IWebshop}$$

The following values and targets are assigned to attributes and references:

$$\text{checkout.parameters} \xrightarrow{references} \text{customerId},$$

$$\text{IWebshop.events} \xrightarrow{references} \text{checkout}$$

C.4 Shared Interface Hierarchy

Figure C.4 shows an example model of the profile *Shared Interface Hierarchy*. Example 12 gives the formal definition of this model.

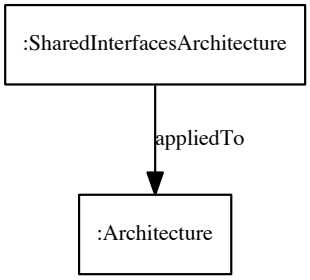


Figure C.4: Example application of the profile *Shared Interface Hierarchy*

Example 12: Example application of the profile *Shared Interface Hierarchy*

The exemplary profile application $M^{Example}_{SharedInterfaceHierarchy}$, that instantiates the profile *Shared Interface Hierarchy*, is formalized as follows. Figure C.4 accompanies the definition as an overview.

$$P = P_{SharedInterfaceHierarchy}$$

$$R = \{M^{Example}_{Kernel}\}$$

The stereotypes are applied as follows:

$$\text{SharedInterfacesArchitecture} \xrightarrow{appliedTo} o_{Architecture}$$

C.5 Scoped Interface Hierarchy

Figure C.5 shows an example model of the profile *Scoped Interface Hierarchy*. Example 13 gives the formal definition of this model.

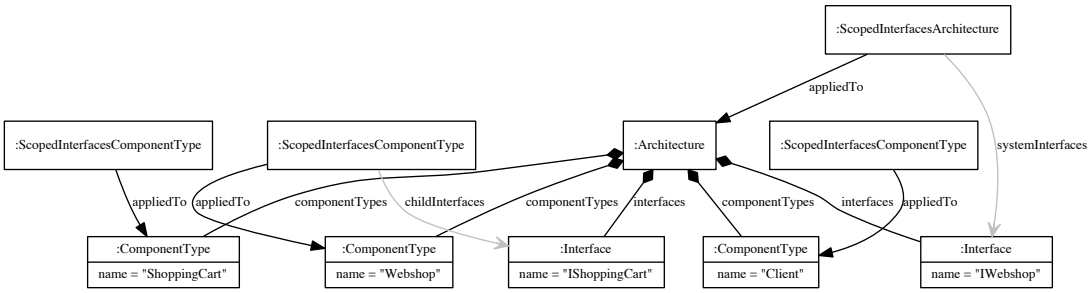


Figure C.5: Example application of the profile *Scoped Interface Hierarchy*

Example 13: Example application of the profile *Scoped Interface Hierarchy*

The exemplary profile application $M^{Example}_{ScopedInterfaceHierarchy}$, that instantiates the profile *Scoped Interface Hierarchy*, is formalized as follows. Figure C.5 accompanies the definition as an overview.

$$P = P_{ScopedInterfaceHierarchy}$$

$$R = \{M^{Example}_{Kernel}\}$$

The stereotypes are applied as follows:

$$\text{ScopedInterfacesArchitecture} \xrightarrow{appliedTo} o_{Architecture},$$

$$\text{ScopedInterfacesComponentType} \xrightarrow{appliedTo} \text{Webshop},$$

$$\text{ScopedInterfacesComponentType} \xrightarrow{appliedTo} \text{Client},$$

$$\text{ScopedInterfacesComponentType} \xrightarrow{appliedTo} \text{ShoppingCart}$$

The following values and targets are assigned to attributes and references:

$$o_{Architecture}.\text{systemInterfaces} \xrightarrow{references} \text{IWebshop},$$

$$\text{Webshop.childInterfaces} \xrightarrow{references} \text{IShoppingCart}$$

C.6 Flat Component Hierarchy

Figure C.6 shows an example model of the profile *Flat Component Hierarchy*. Example 14 gives the formal definition of this model.

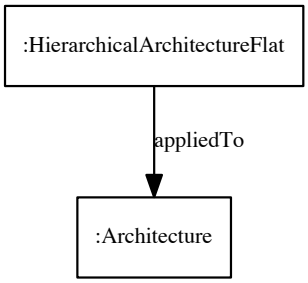


Figure C.6: Example application of the profile *Flat Component Hierarchy*

Example 14: Example application of the profile *Flat Component Hierarchy*

The exemplary profile application $M^{Example}_{FlatComponentHierarchy}$, that instantiates the profile *Flat Component Hierarchy*, is formalized as follows. Figure C.6 accompanies the definition as an overview.

$$P = P_{FlatComponentHierarchy}$$

$$R = \{M^{Example}_{Kernel}\}$$

The stereotypes are applied as follows:

$$\text{HierarchicalArchitectureFlat} \xrightarrow{appliedTo} o_{Architecture}$$

C.7 Scoped Component Hierarchy

Figure C.7 shows an example model of the profile *Scoped Component Hierarchy*. Example 15 gives the formal definition of this model.

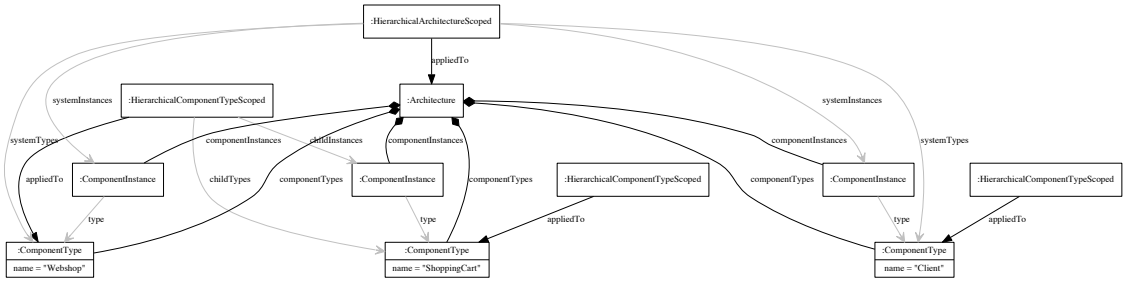


Figure C.7: Example application of the profile *Scoped Component Hierarchy*

Example 15: Example application of the profile *Scoped Component Hierarchy*

The exemplary profile application $M^{Example}_{ScopedComponentHierarchy}$, that instantiates the profile *Scoped Component Hierarchy*, is formalized as follows. Figure C.7 accompanies the definition as an overview.

$$P = P_{ScopedComponentHierarchy}$$

$$R = \{M^{Example}_{Kernel}\}$$

The stereotypes are applied as follows:

$$\text{HierarchicalArchitectureScoped} \xrightarrow{appliedTo} o_{Architecture},$$

$$\text{HierarchicalComponentTypeScoped} \xrightarrow{appliedTo} \text{Webshop},$$

$$\text{HierarchicalComponentTypeScoped} \xrightarrow{appliedTo} \text{Client},$$

$$\text{HierarchicalComponentTypeScoped} \xrightarrow{appliedTo} \text{ShoppingCart}$$

The following values and targets are assigned to attributes and references:

$$o_{Architecture}.\text{systemTypes} \xrightarrow{references} \text{Webshop},$$

$$o_{Architecture}.\text{systemTypes} \xrightarrow{references} \text{Client},$$

$$o_{Architecture}.\text{systemInstances} \xrightarrow{references} o^{Webshop}_{ComponentInstance},$$

$$o_{Architecture}.\text{systemInstances} \xrightarrow{references} o^{Client}_{ComponentInstance},$$

$$\text{Webshop.childInstances} \xrightarrow{references} o^{ShoppingCart}_{ComponentInstance},$$
$$\text{Webshop.childTypes} \xrightarrow{references} \text{ShoppingCart}$$

C.8 Shared Context Component Hierarchy

Figure C.8 shows an example model of the profile *Shared Context Component Hierarchy*. Example 16 gives the formal definition of this model.

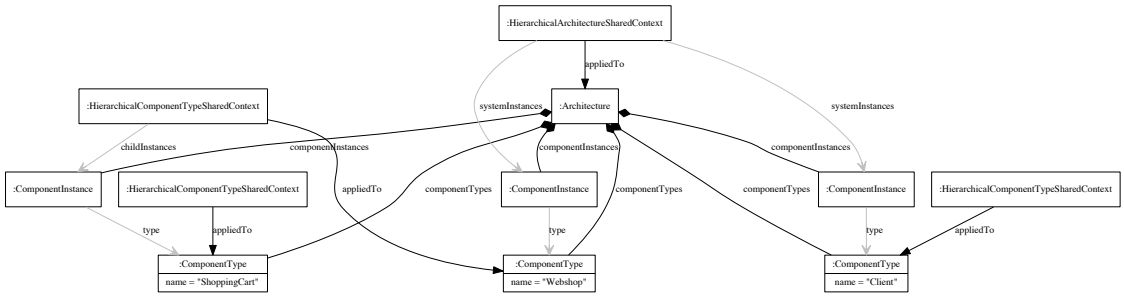


Figure C.8: Example application of the profile *Shared Context Component Hierarchy*

Example 16: Example application of the profile *Shared Context Component Hierarchy*

The exemplary profile application $M^{Example}_{SharedContextComponentHierarchy}$, that instantiates the profile *Shared Context Component Hierarchy*, is formalized as follows. Figure C.8 accompanies the definition as an overview.

$$P = P_{SharedContextComponentHierarchy}$$

$$R = \{M^{Example}_{Kernel}\}$$

The stereotypes are applied as follows:

$$\text{HierarchicalArchitectureSharedContext} \xrightarrow{appliedTo} o_{Architecture},$$

$$\text{HierarchicalComponentTypeSharedContext} \xrightarrow{appliedTo} \text{Webshop},$$

$$\text{HierarchicalComponentTypeSharedContext} \xrightarrow{appliedTo} \text{Client},$$

$$\text{HierarchicalComponentTypeSharedContext} \xrightarrow{appliedTo} \text{ShoppingCart}$$

The following values and targets are assigned to attributes and references:

$$o_{Architecture}.\text{systemInstances} \xrightarrow{references} o^{Webshop}_{ComponentInstance},$$

$$o_{Architecture}.\text{systemInstances} \xrightarrow{references} o^{Client}_{ComponentInstance},$$

$$\text{Webshop.childInstances} \xrightarrow{references} O^{ShoppingCart}_{ComponentInstance}$$

C.8.1 Dependency: Example model of the IAL Kernel

Figure C.9 shows an example model that uses the IAL *Kernel*. Example 17 gives the formal definition of this model.

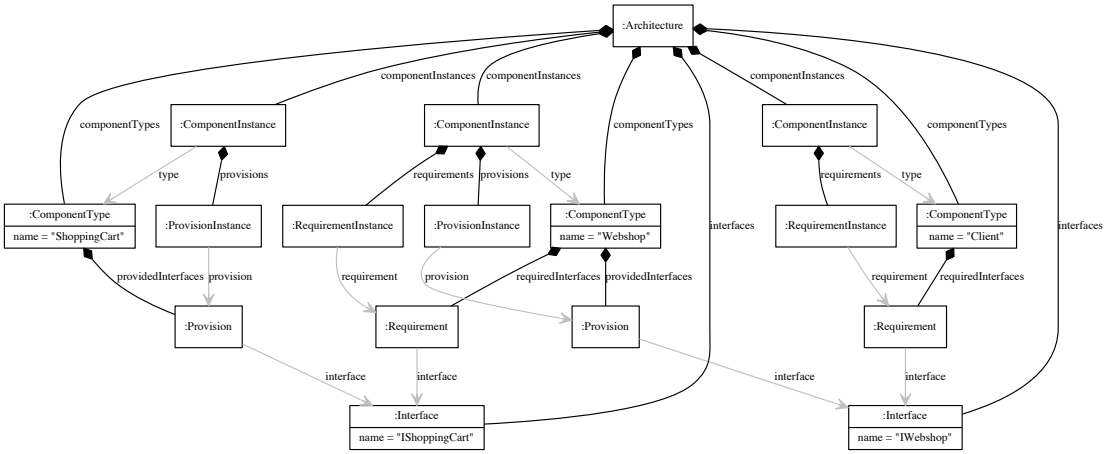


Figure C.9: Example model which instantiates the IAL Kernel

Example 17: Example dependency model of the IAL Kernel

The exemplary model $M_{Kernel}^{Dependency-ComponentHierarchyShared}$, that instantiates the IAL *Kernel*, is formalized as follows. Figure C.9 accompanies the definition as an overview.

$$L_{Meta} = M_{Meta}^{Kernel}$$

$$\begin{aligned} O := \{ & o_{Architecture}, \\ & o_{IWebshop}, \\ & o_{IShoppingCart}, \\ & o_{Webshop}, \\ & o_{Requirement}, \\ & o_{Provision}, \\ & o_{Client}, \\ & o_{Requirement}, \\ & o_{ShoppingCart}, \\ & o_{Provision}, \\ & o_{ComponentInstance}, \end{aligned}$$

$$
\begin{aligned}
&o_{RequirementInstance},\\
&o_{ProvisionInstance},\\
&o_{ComponentInstance},\\
&o_{RequirementInstance},\\
&o_{ComponentInstance},\\
&o_{ProvisionInstance}\}
\end{aligned}
$$

The elements are named as follows:

$$
\begin{aligned}
&name(o_{IWebshop}) = \text{IWebshop},\\
&name(o_{IShoppingCart}) = \text{IShoppingCart},\\
&name(o_{Webshop}) = \text{Webshop},\\
&name(o_{Client}) = \text{Client},\\
&name(o_{ShoppingCart}) = \text{ShoppingCart}
\end{aligned}
$$

The following values and targets are assigned to attributes and references:

$$
\begin{aligned}
&\text{Architecture.interfaces} \xrightarrow{references} \text{IWebshop},\\
&\text{Architecture.interfaces} \xrightarrow{references} \text{IShoppingCart},\\
&\text{Requirement.interface} \xrightarrow{references} \text{IShoppingCart},\\
&\text{Webshop.requiredInterfaces} \xrightarrow{references} \text{Requirement},\\
&\text{Provision.interface} \xrightarrow{references} \text{IWebshop},\\
&\text{Webshop.providedInterfaces} \xrightarrow{references} \text{Provision},\\
&\text{Architecture.componentTypes} \xrightarrow{references} \text{Webshop},\\
&\text{Requirement.interface} \xrightarrow{references} \text{IWebshop},\\
&\text{Client.requiredInterfaces} \xrightarrow{references} \text{Requirement},\\
&\text{Architecture.componentTypes} \xrightarrow{references} \text{Client},\\
&\text{Provision.interface} \xrightarrow{references} \text{IShoppingCart},\\
&\text{ShoppingCart.providedInterfaces} \xrightarrow{references} \text{Provision},\\
&\text{Architecture.componentTypes} \xrightarrow{references} \text{ShoppingCart},\\
&\text{RequirementInstance.requirement} \xrightarrow{references} \text{Requirement},\\
&\text{ComponentInstance.requirements} \xrightarrow{references} \text{RequirementInstance},\\
&\text{ProvisionInstance.provision} \xrightarrow{references} \text{Provision},\\
&\text{ComponentInstance.provisions} \xrightarrow{references} \text{ProvisionInstance},
\end{aligned}
$$

$$\text{ComponentInstance.type} \xrightarrow{references} \text{Webshop},$$
$$\text{Architecture.componentInstances} \xrightarrow{references} \text{ComponentInstance},$$
$$\text{RequirementInstance.requirement} \xrightarrow{references} \text{Requirement},$$
$$\text{ComponentInstance.requirements} \xrightarrow{references} \text{RequirementInstance},$$
$$\text{ComponentInstance.type} \xrightarrow{references} \text{Client},$$
$$\text{Architecture.componentInstances} \xrightarrow{references} \text{ComponentInstance},$$
$$\text{ProvisionInstance.provision} \xrightarrow{references} \text{Provision},$$
$$\text{ComponentInstance.provisions} \xrightarrow{references} \text{ProvisionInstance},$$
$$\text{ComponentInstance.type} \xrightarrow{references} \text{ShoppingCart},$$
$$\text{Architecture.componentInstances} \xrightarrow{references} \text{ComponentInstance}$$

C.9 Fixed Component Instantiation

Figure C.10 shows an example model of the profile *Fixed Component Instantiation*. Example 18 gives the formal definition of this model.

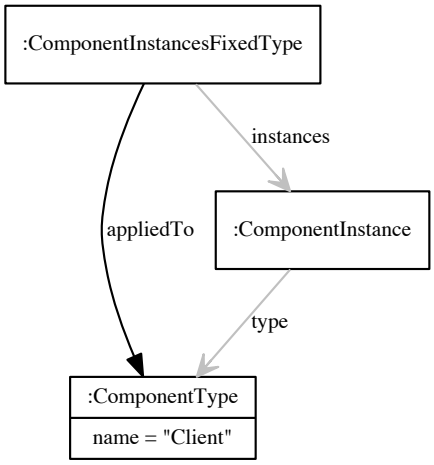


Figure C.10: Example application of the profile *Fixed Component Instantiation*

Example 18: Example application of the profile *Fixed Component Instantiation*

The exemplary profile application $M^{Example}_{FixedComponentInstantiation}$, that instantiates the profile *Fixed Component Instantiation*, is formalized as follows. Figure C.10 accompanies the definition as an overview.

$$P = P_{FixedComponentInstantiation}$$

$$R = \{M^{Example}_{Kernel}\}$$

The stereotypes are applied as follows:

$$\text{ComponentInstancesFixedType} \xrightarrow{appliedTo} \text{Client}$$

The following values and targets are assigned to attributes and references:

$$\text{Client.instances} \xrightarrow{references} o^{Client}_{ComponentInstance}$$

C.10 Per Session Component Instantiation

Figure C.11 shows an example model of the profile *Per Session Component Instantiation*. Example 19 gives the formal definition of this model.

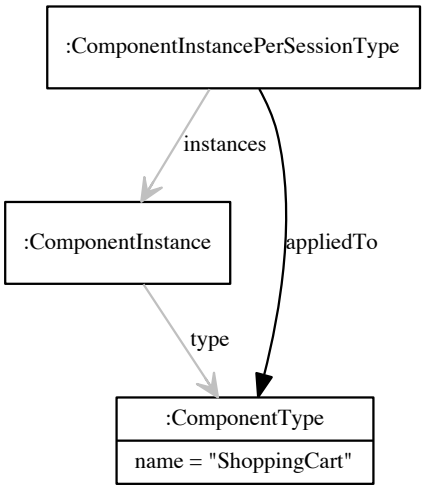


Figure C.11: Example application of the profile *Per Session Component Instantiation*

Example 19: Example application of the profile *Per Session Component Instantiation*

The exemplary profile application $M^{Example}_{PerSessionComponentInstantiation}$, that instantiates the profile *Per Session Component Instantiation*, is formalized as follows. Figure C.11 accompanies the definition as an overview.

$$P = P_{PerSessionComponentInstantiation}$$

$$R = \{M^{Example}_{Kernel}\}$$

The stereotypes are applied as follows:

$$\text{ComponentInstancePerSessionType} \xrightarrow{appliedTo} \text{ShoppingCart}$$

The following values and targets are assigned to attributes and references:

$$\text{ShoppingCart.instances} \xrightarrow{references} o^{ShoppingCart}_{ComponentInstance}$$

C.11 Pooled Component Instantiation

Figure C.12 shows an example model of the profile *Pooled Component Instantiation*. Example 20 gives the formal definition of this model.

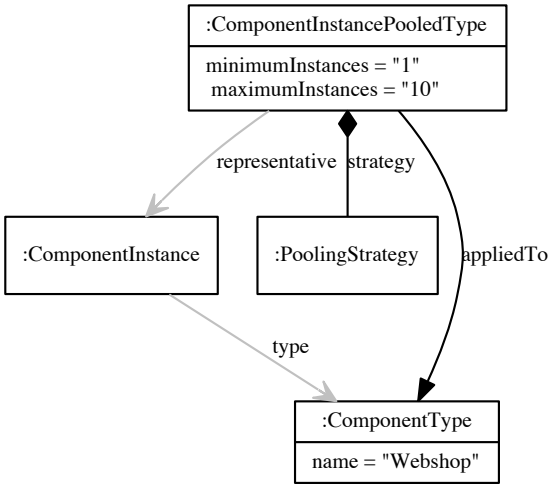


Figure C.12: Example application of the profile *Pooled Component Instantiation*

Example 20: Example application of the profile *Pooled Component Instantiation*

The exemplary profile application $M^{Example}_{PooledComponentInstantiation}$, that instantiates the profile *Pooled Component Instantiation*, is formalized as follows. Figure C.12 accompanies the definition as an overview.

$$P = P_{PooledComponentInstantiation}$$

$$R = \{M^{Example}_{Kernel}\}$$

$$O := \{o_{PoolingStrategy}\}$$

The elements are named as follows:

$$name(o_{PoolingStrategy}) = \text{PoolingStrategy}$$

The stereotypes are applied as follows:

$$\text{ComponentInstancePooledType} \xrightarrow{appliedTo} \text{Webshop}$$

The following values and targets are assigned to attributes and references:

$$\text{Webshop.minimumInstances} \xrightarrow{hasValue} 1,$$
$$\text{Webshop.maximumInstances} \xrightarrow{hasValue} 10,$$
$$\text{Webshop.representative} \xrightarrow{references} o^{Webshop}_{ComponentInstance},$$
$$\text{Webshop.strategy} \xrightarrow{references} \text{PoolingStrategy}$$

C.12 Stateful Components

Figure C.13 shows an example model of the profile *Stateful Components*. Example 21 gives the formal definition of this model.

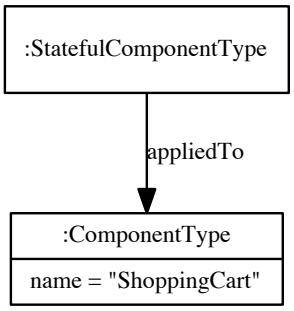


Figure C.13: Example application of the profile *Stateful Components*

Example 21: Example application of the profile *Stateful Components*

The exemplary profile application $M^{Example}_{StatefulComponents}$, that instantiates the profile *Stateful Components*, is formalized as follows. Figure C.13 accompanies the definition as an overview.

$$P = P_{StatefulComponents}$$

$$R = \{M^{Example}_{Kernel}\}$$

The stereotypes are applied as follows:

$$\text{StatefulComponentType} \xrightarrow{appliedTo} \text{ShoppingCart}$$

C.13 Stateless Components

Figure C.14 shows an example model of the profile *Stateless Components*. Example 22 gives the formal definition of this model.

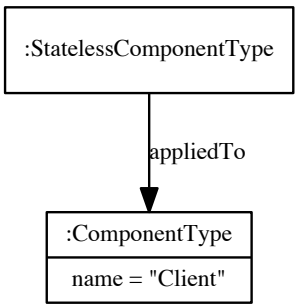


Figure C.14: Example application of the profile *Stateless Components*

Example 22: Example application of the profile *Stateless Components*

The exemplary profile application $M^{Example}_{StatelessComponents}$, that instantiates the profile *Stateless Components*, is formalized as follows. Figure C.14 accompanies the definition as an overview.

$$P = P_{StatelessComponents}$$

$$R = \{M^{Example}_{Kernel}\}$$

The stereotypes are applied as follows:

$$\text{StatelessComponentType} \xrightarrow{appliedTo} \text{Client}$$

C.14 State Machine

Figure C.15 shows an example model of the profile *State Machine*. Example 23 gives the formal definition of this model.

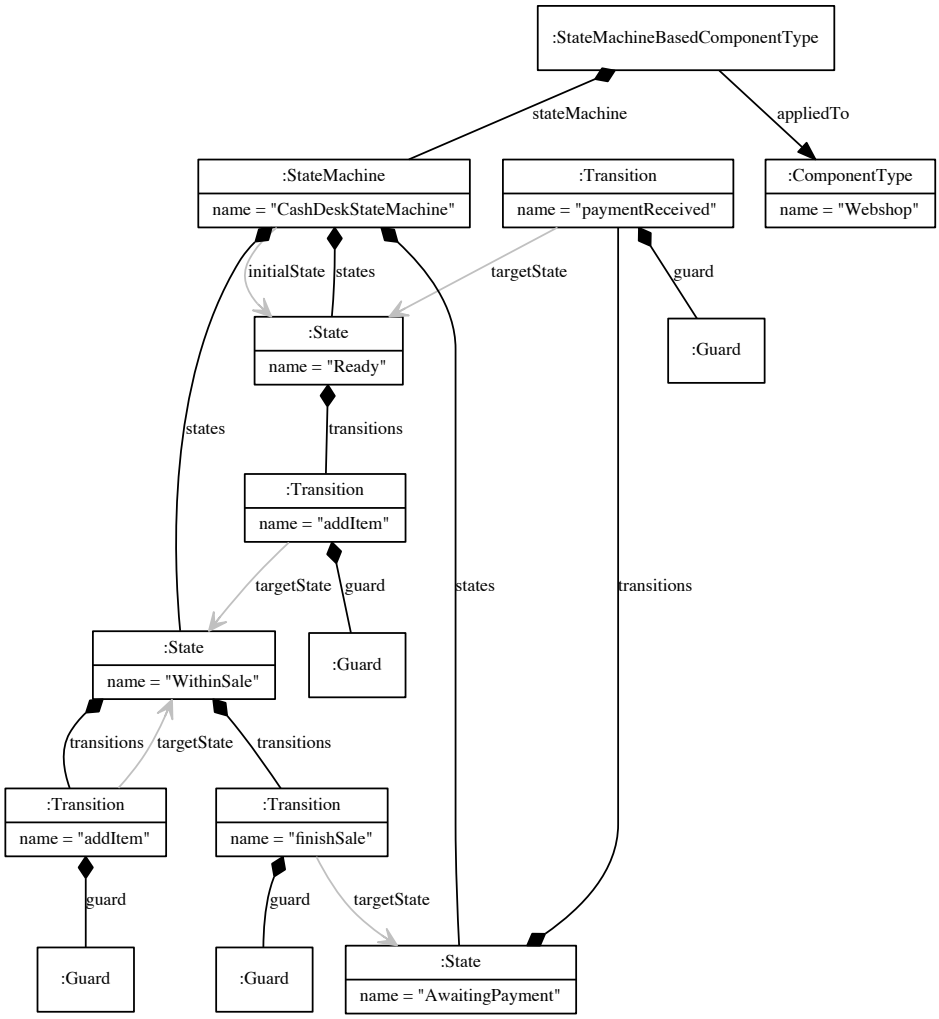


Figure C.15: Example application of the profile *State Machine*

Example 23: Example application of the profile *State Machine*

The exemplary profile application $M^{Example}_{StateMachine}$, that instantiates the profile *State Machine*, is formalized as follows. Figure C.15 accompanies the definition as an overview.

$$P = P_{Namespace}$$

$$R = \{M^{Example}_{Kernel}\}$$

$$\begin{aligned} O := \{&o_{CashDeskStateMachine}, o_{Ready}, o_{WithinSale}, o_{AwaitingPayment},\\ &o^{WithinSale}_{addItem}, o^{Ready}_{addItem}, o_{paymentReceived}, o_{finishSale},\\ &o^{addItem-WithinSale}_{Guard}, o^{addItem-Ready}_{Guard}, o^{paymentReceived}_{Guard}, o^{finishSale}_{Guard}\} \end{aligned}$$

The elements are named as follows:

$$\begin{aligned} &name(o_{CashDeskStateMachine}) = \text{CashDeskStateMachine},\\ &name(o_{Ready}) = \text{Ready},\\ &name(o_{WithinSale}) = \text{WithinSale},\\ &name(o_{AwaitingPayment}) = \text{AwaitingPayment},\\ &name(o^{WithinSale}_{addItem}) = \text{addItem},\\ &name(o^{Ready}_{addItem}) = \text{addItem},\\ &name(o_{finishSale}) = \text{finishSale},\\ &name(o_{paymentReceived}) = \text{paymentReceived} \end{aligned}$$

The stereotypes are applied as follows:

$$\text{StateMachineBasedComponentType} \xrightarrow{appliedTo} \text{Webshop}$$

The following values and targets are assigned to attributes and references:

$$\begin{aligned} &\text{Webshop.stateMachine} \xrightarrow{references} \text{CashDeskStateMachine},\\ &\text{CashDeskStateMachine.states} \xrightarrow{references} \text{Ready},\\ &\text{CashDeskStateMachine.states} \xrightarrow{references} \text{WithinSale},\\ &\text{CashDeskStateMachine.states} \xrightarrow{references} \text{AwaitingPayment},\\ &\text{CashDeskStateMachine.initialState} \xrightarrow{references} \text{Ready},\\ &\text{Ready.transitions} \xrightarrow{references} o^{Ready}_{addItem},\\ &\text{WithinSale.transitions} \xrightarrow{references} o^{WithinSale}_{addItem},\\ &\text{WithinSale.transitions} \xrightarrow{references} \text{finishSale}, \end{aligned}$$

$\text{AwaitingPayment.transitions} \xrightarrow{references} \text{paymentReceived},$

$o^{Ready}_{addItem}.\text{targetState} \xrightarrow{references} \text{WithinSale},$

$o^{WithinSale}_{addItem}.\text{targetState} \xrightarrow{references} \text{WithinSale},$

$\text{finishSale.targetState} \xrightarrow{references} \text{AwaitingPayment},$

$\text{paymentReceived.targetState} \xrightarrow{references} \text{Ready},$

$o^{Ready}_{addItem}.\text{guard} \xrightarrow{references} o^{addItem-Ready}_{Guard},$

$o^{WithinSale}_{addItem}.\text{guard} o^{addItem-WithinSale}_{Guard},$

$\text{paymentReceived.guard} \xrightarrow{references} o^{paymentReceived}_{Guard},$

$\text{finishSale.guard} \xrightarrow{references} o^{finishSale}_{Guard}$

C.15 Connector

Figure C.16 shows an example model of the profile *Connector*. Example 24 gives the formal definition of this model.

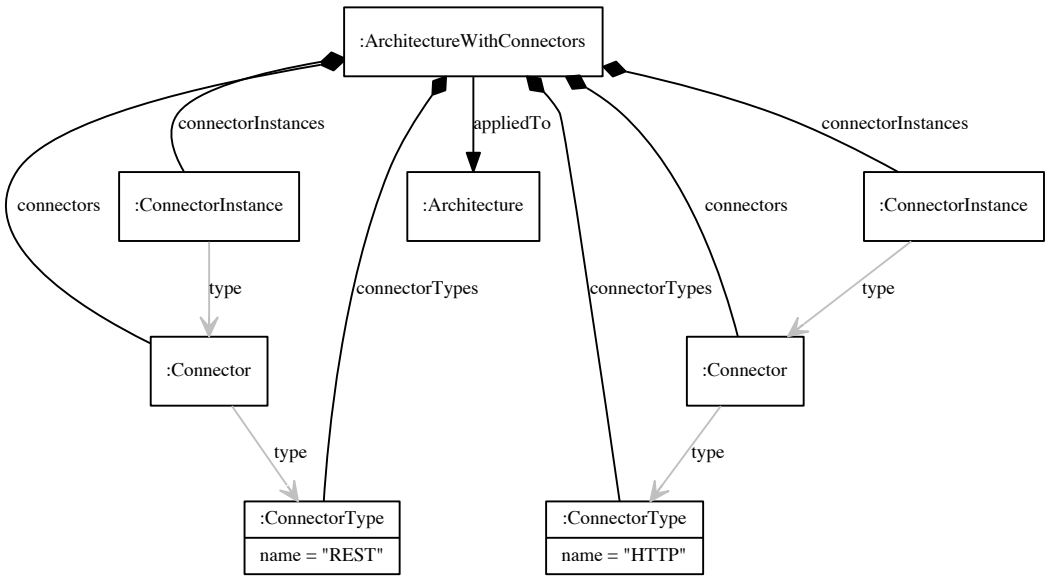


Figure C.16: Example application of the profile *Connector*

Example 24: Example application of the profile *Connector*

The exemplary profile application $M^{Example}_{Connector}$, that instantiates the profile *Connector*, is formalized as follows. Figure C.16 accompanies the definition as an overview.

$$P = P_{Connector}$$

$$R = \{M^{Example}_{Kernel}\}$$

$$O := \{o_{HTTP}, o_{REST}, o^{HTTP}_{Connector}, o^{REST}_{Connector}, o^{HTTP}_{ConnectorInstance}, o^{REST}_{ConnectorInstance}\}$$

The elements are named as follows:

$$name(o_{HTTP}) = \text{HTTP},$$
$$name(o_{REST}) = \text{REST}$$

The stereotypes are applied as follows:

$$\text{ArchitectureWithConnectors} \xrightarrow{appliedTo} o_{Architecture}$$

The following values and targets are assigned to attributes and references:

$$o_{Architecture}.\text{connectorTypes} \xrightarrow{references} \text{HTTP},$$

$$o_{Architecture}.\text{connectorTypes} \xrightarrow{references} \text{REST},$$

$$o^{HTTP}_{Connector}.\text{type} \xrightarrow{references} \text{HTTP},$$

$$o^{REST}_{Connector}.\text{type} \xrightarrow{references} \text{REST},$$

$$o_{Architecture}.\text{connectors} \xrightarrow{references} o^{HTTP}_{Connector},$$

$$o_{Architecture}.\text{connectors} \xrightarrow{references} o^{REST}_{Connector},$$

$$o^{HTTP}_{ConnectorInstance}.\text{type} \xrightarrow{references} o^{HTTP}_{Connector},$$

$$o^{REST}_{ConnectorInstance}.\text{type} \xrightarrow{references} o^{REST}_{Connector},$$

$$o_{Architecture}.\text{connectorInstances} \xrightarrow{references} o^{HTTP}_{ConnectorInstance},$$

$$o_{Architecture}.\text{connectorInstances} \xrightarrow{references} o^{REST}_{ConnectorInstance}$$

C.16 Operation Call Connector

Figure C.17 shows an example model of the profile *Operation Call Connector*. Example 25 gives the formal definition of this model.

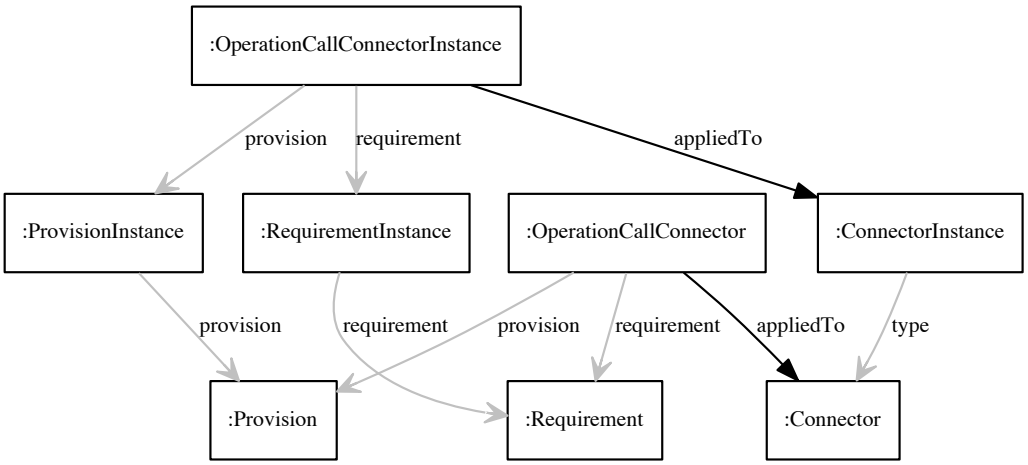


Figure C.17: Example application of the profile *Operation Call Connector*

Example 25: Example application of the profile *Operation Call Connector*

The exemplary profile application $M^{Example}_{OperationCallConnector}$, that instantiates the profile *Operation Call Connector*, is formalized as follows. Figure C.17 accompanies the definition as an overview.

$$P = P_{OperationCallConnector}$$

$$R = \{M^{Example}_{Connector}, M^{Example}_{Kernel}\}$$

The stereotypes are applied as follows:

$$\text{OperationCallConnector} \xrightarrow{appliedTo} o^{REST}_{Connector},$$

$$\text{OperationCallConnectorInstance} \xrightarrow{appliedTo} o^{REST}_{ConnectorInstance}$$

The following values and targets are assigned to attributes and references:

$$o^{REST}_{Connector}.\text{requirement} \xrightarrow{references} o^{Client}_{Requirement},$$

$o^{REST}_{Connector}$.provision $\xrightarrow{references}$ $o^{Webshop}_{Provision}$,

$o^{REST}_{ConnectorInstance}$.requirement $\xrightarrow{references}$ $o^{Client}_{RequirementInstance}$,

$o^{REST}_{ConnectorInstance}$.provision $\xrightarrow{references}$ $o^{Webshop}_{ProvisionInstance}$

C.17 Event Dispatcher Connector

Figure C.18 shows an example model of the profile *Event Dispatcher Connector*. Example 26 gives the formal definition of this model.

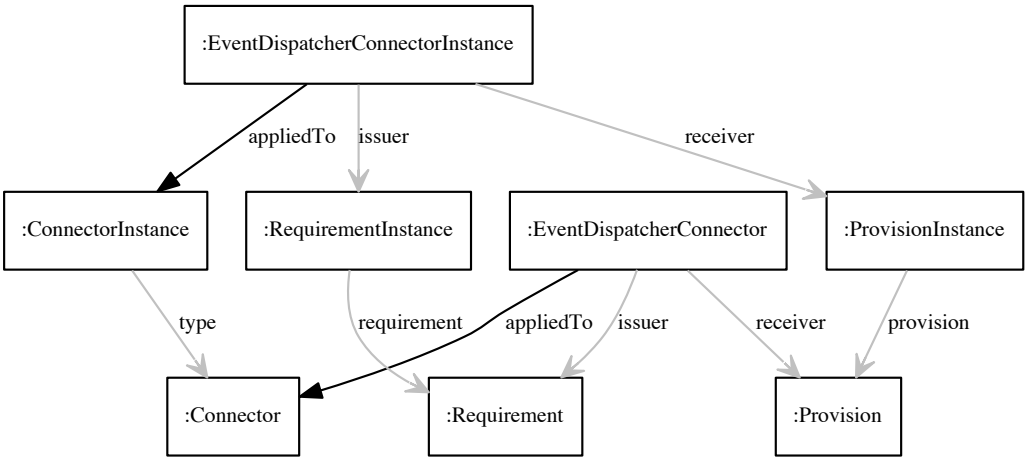


Figure C.18: Example application of the profile *Event Dispatcher Connector*

Example 26: Example application of the profile *Event Dispatcher Connector*

The exemplary profile application $M^{Example}_{EventDispatcherConnector}$, that instantiates the profile *Event Dispatcher Connector*, is formalized as follows. Figure C.18 accompanies the definition as an overview.

$$P = P_{EventDispatcherConnector}$$
$$R = \{M^{Example}_{Connector}, M^{Example}_{Kernel}\}$$

The stereotypes are applied as follows:

$$\text{EventDispatcherConnector} \xrightarrow{appliedTo} o^{REST}_{Connector},$$
$$\text{EventDispatcherConnectorInstance} \xrightarrow{appliedTo} o^{REST}_{ConnectorInstance}$$

The following values and targets are assigned to attributes and references:

$$o^{REST}_{Connector}.\text{receiver} \xrightarrow{references} o^{Webshop}_{Provision},$$

$$o^{REST}_{Connector}.\text{issuer} \xrightarrow{references} o^{Client}_{Requirement},$$
$$o^{REST}_{ConnectorInstance}.\text{receiver} \xrightarrow{references} o^{Webshop}_{ProvisionInstance},$$
$$o^{REST}_{ConnectorInstance}.\text{issuer} \xrightarrow{references} o^{Client}_{RequirementInstance}$$

C.18 Delegation Connector

Figure C.19 shows an example model of the profile *Delegation Connector*. Example 27 gives the formal definition of this model.

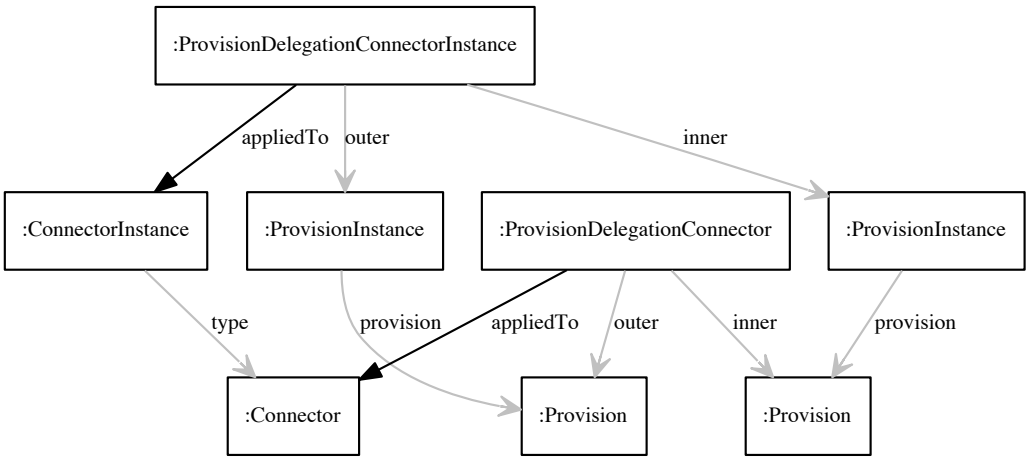


Figure C.19: Example application of the profile *Delegation Connector*

Example 27: Example application of the profile *Delegation Connector*

The exemplary profile application $M^{Example}_{DelegationConnector}$, that instantiates the profile *Delegation Connector*, is formalized as follows. Figure C.19 accompanies the definition as an overview.

$$P = P_{DelegationConnector}$$

$$R = \{M^{Example}_{Connector}, M^{Example}_{Kernel}\}$$

The stereotypes are applied as follows:

$$\text{ProvisionDelegationConnector} \xrightarrow{appliedTo} o^{HTTP}_{Connector},$$

$$\text{ProvisionDelegationConnectorInstance} \xrightarrow{appliedTo} o^{HTTP}_{ConnectorInstance}$$

The following values and targets are assigned to attributes and references:

$$o^{HTTP}_{Connector}.\text{outer} \xrightarrow{references} o^{Webshop}_{Provision},$$

$$o^{HTTP}_{Connector}.\text{inner} \xrightarrow{references} o^{ShoppingCart}_{Provision},$$

$$o^{HTTP}_{ConnectorInstance}.\text{outer} \xrightarrow{references} o^{Webshop}_{ProvisionInstance},$$
$$o^{HTTP}_{ConnectorInstance}.\text{inner} \xrightarrow{references} o^{ShoppingCart}_{ProvisionInstance}$$

C.19 Datatypes Common

Figure C.20 shows an example model of the profile *Datatypes Common*. Example 28 gives the formal definition of this model.

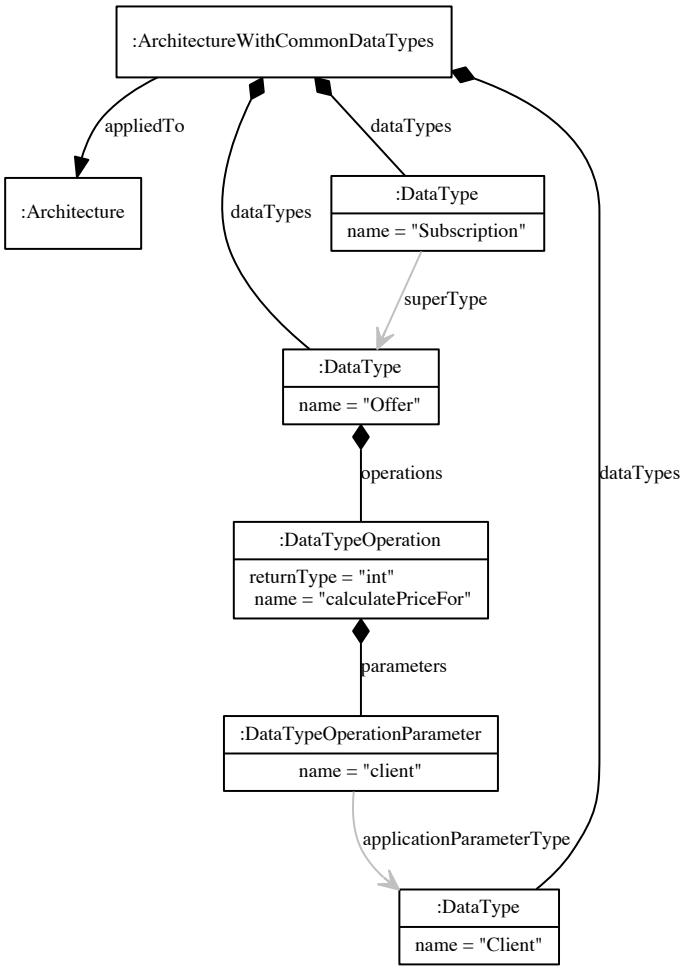


Figure C.20: Example application of the profile *Datatypes Common*

Example 28: Example application of the profile *Datatypes Common*

The exemplary profile application $M^{Example}_{DatatypesCommon}$, that instantiates the profile *Datatypes Common*, is formalized as follows. Figure C.20 accompanies the definition as an overview.

$$P = P_{DatatypesCommon}$$
$$R = \{M^{Example}_{Kernel}\}$$

$$O := \{o_{Offer}, o_{calculatePriceFor}, o_{client}, o_{Client}, o_{Subscription}\}$$

The elements are named as follows:

$$name(o_{Offer}) = \text{Offer},$$
$$name(o_{calculatePriceFor}) = \text{calculatePriceFor},$$
$$name(o_{client}) = \text{client},$$
$$name(o_{Client}) = \text{Client},$$
$$name(o_{Subscription}) = \text{Subscription}$$

The stereotypes are applied as follows:

$$\text{ArchitectureWithCommonDataTypes} \xrightarrow{appliedTo} o_{Architecture}$$

The following values and targets are assigned to attributes and references:

$$o_{Architecture}.\text{dataTypes} \xrightarrow{references} \text{Offer},$$
$$o_{Architecture}.\text{dataTypes} \xrightarrow{references} \text{Subscription},$$
$$o_{Architecture}.\text{dataTypes} \xrightarrow{references} \text{Client},$$
$$\text{Subscription.superType} \xrightarrow{references} \text{Offer},$$
$$\text{Offer.operations} \xrightarrow{references} \text{calculatePriceFor},$$
$$\text{calculatePriceFor.returnType} \xrightarrow{hasValue} \text{int},$$
$$\text{calculatePriceFor.parameters} \xrightarrow{references} \text{client},$$
$$\text{client.applicationParameterType} \xrightarrow{references} \text{Client},$$

C.20 Datatypes Operations

Figure C.21 shows an example model of the profile *Datatypes Operations*. Example 29 gives the formal definition of this model.

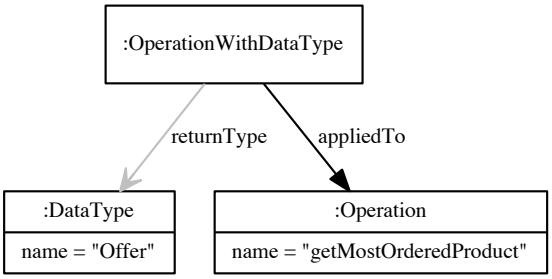


Figure C.21: Example application of the profile *Datatypes Operations*

Example 29: Example application of the profile *Datatypes Operations*

The exemplary profile application $M^{Example}_{DatatypesOperations}$, that instantiates the profile *Datatypes Operations*, is formalized as follows. Figure C.21 accompanies the definition as an overview.

$$P = P_{DatatypesOperations}$$

$$R = \{M^{Dependency-DatatypeOperations}_{InterfaceTypeOperations}, M^{Example}_{DatatypesCommon}\}$$

The stereotypes are applied as follows:

$$\text{OperationWithDataType} \xrightarrow{appliedTo} \text{getMostOrderedProduct}$$

The following values and targets are assigned to attributes and references:

$$\text{getMostOrderedProduct.returnType} \xrightarrow{references} \text{Offer}$$

C.20.1 Dependency: Example profile application of the profile *Operation Interfaces*

Figure C.22 shows an example model that uses the profile *Interface Type Operations*. Example 30 gives the formal definition of this model.

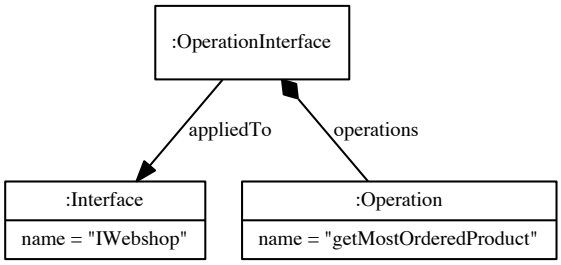


Figure C.22: Example profile application which instantiates the profile *Operation Interfaces*

Example 30: Example dependency profile application of the profile *Operation Interfaces*

The exemplary profile application $M^{Dependency-DatatypeOperations}_{InterfaceTypeOperations}$, that instantiates the profile *Interface Type Operations*, is formalized as follows. Figure C.22 accompanies the definition as an overview.

$$P = P_{OperationInterfaces}$$

$$R = \{M^{Example}_{Kernel}\}$$

$$O := \{o_{getMostOrderedProduct}\}$$

The elements are named as follows:

$$name(o_{getMostOrderedProduct}) = \text{getMostOrderedProduct}$$

The stereotypes are applied as follows:

$$\text{OperationInterface} \xrightarrow{appliedTo} \text{IWebshop}$$

The following values and targets are assigned to attributes and references:

$$\text{IWebshop.operations} \xrightarrow{references} \text{getMostOrderedProduct}$$

C.21 Datatypes Events

Figure C.23 shows an example model of the profile *Datatypes Events*. Example 31 gives the formal definition of this model.

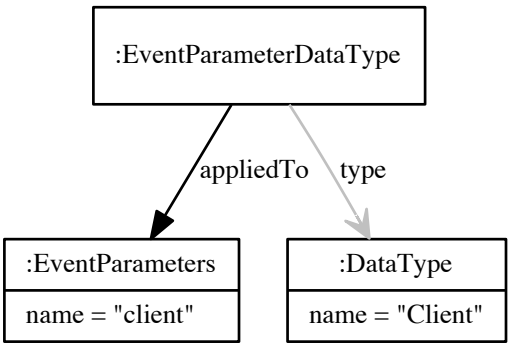


Figure C.23: Example application of the profile *Datatypes Events*

Example 31: Example application of the profile *Datatypes Events*

The exemplary profile application $M^{Example}_{DatatypesEvents}$, that instantiates the profile *Datatypes Events*, is formalized as follows. Figure C.23 accompanies the definition as an overview.

$$P = P_{DatatypesEvents}$$

$$R = \{M^{Dependency-DatatypeEvents}_{InterfaceTypeEvents}, M^{Example}_{DatatypesCommon}\}$$

The stereotypes are applied as follows:

$$\text{EventParameterDataType} \xrightarrow{appliedTo} \text{client}$$

The following values and targets are assigned to attributes and references:

$$\text{client.type} \xrightarrow{references} \text{Client}$$

C.21.1 Dependency: Example profile application of the profile *Event Interfaces*

Figure C.24 shows an example model that uses the profile *Interface Type Events*. Example 32 gives the formal definition of this model.

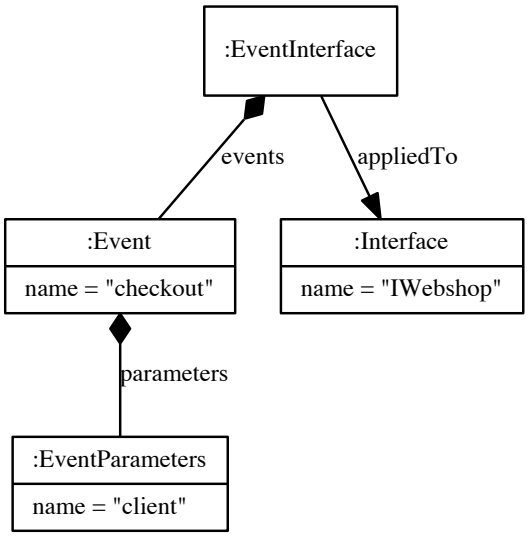


Figure C.24: Example profile application which instantiates the profile *Event Interfaces*

Example 32: Example dependency profile application of the profile *Event Interfaces*

The exemplary profile application $M_{InterfaceTypeEvents}^{Dependency-DatatypeEvents}$, that instantiates the profile *Interface Type Events*, is formalized as follows. Figure C.24 accompanies the definition as an overview.

$$P = P_{EventInterfaces}$$
$$R = \{M_{Kernel}^{Example}\}$$
$$O := \{o_{checkout}, o_{client}\}$$

The elements are named as follows:

$$name(o_{checkout}) = \text{checkout},$$
$$name(o_{client}) = \text{client}$$

The stereotypes are applied as follows:

$$\text{EventInterface} \xrightarrow{appliedTo} \text{IWebshop}$$

The following values and targets are assigned to attributes and references:

$$\text{checkout.parameters} \xrightarrow{references} \text{client},$$

$$\text{IWebshop.events} \xrightarrow{references} \text{checkout}$$

C.22 Deployment

Figure C.25 shows an example model of the profile *Deployment*. Example 33 gives the formal definition of this model.

Example 33: Example application of the profile *Deployment*

The exemplary profile application $M^{Example}_{Deployment}$, that instantiates the profile *Deployment*, is formalized as follows. Figure C.25 accompanies the definition as an overview.

$$P = P_{Deployment}$$

$$R = \{M^{Example}_{Kernel}\}$$

$$\begin{aligned} O := \{&o^{Webshop}_{DeploymentFragment}, o^{Client}_{DeploymentFragment}, o^{ShoppingCart}_{DeploymentFragment}, \\ &o^{Server1}_{ResourceContainer}, o^{Server2}_{ResourceContainer}, o^{Server3}_{ResourceContainer}, \\ &o^{1}_{AllocationContext}, o^{2}_{AllocationContext}, o^{3}_{AllocationContext}\} \end{aligned}$$

The elements are named as follows:

$$\begin{aligned} &name(o_{Webshop}) = \text{Webshop}, \\ &name(o_{Client}) = \text{Client}, \\ &name(o_{ShoppingCart}) = \text{ShoppingCart} \end{aligned}$$

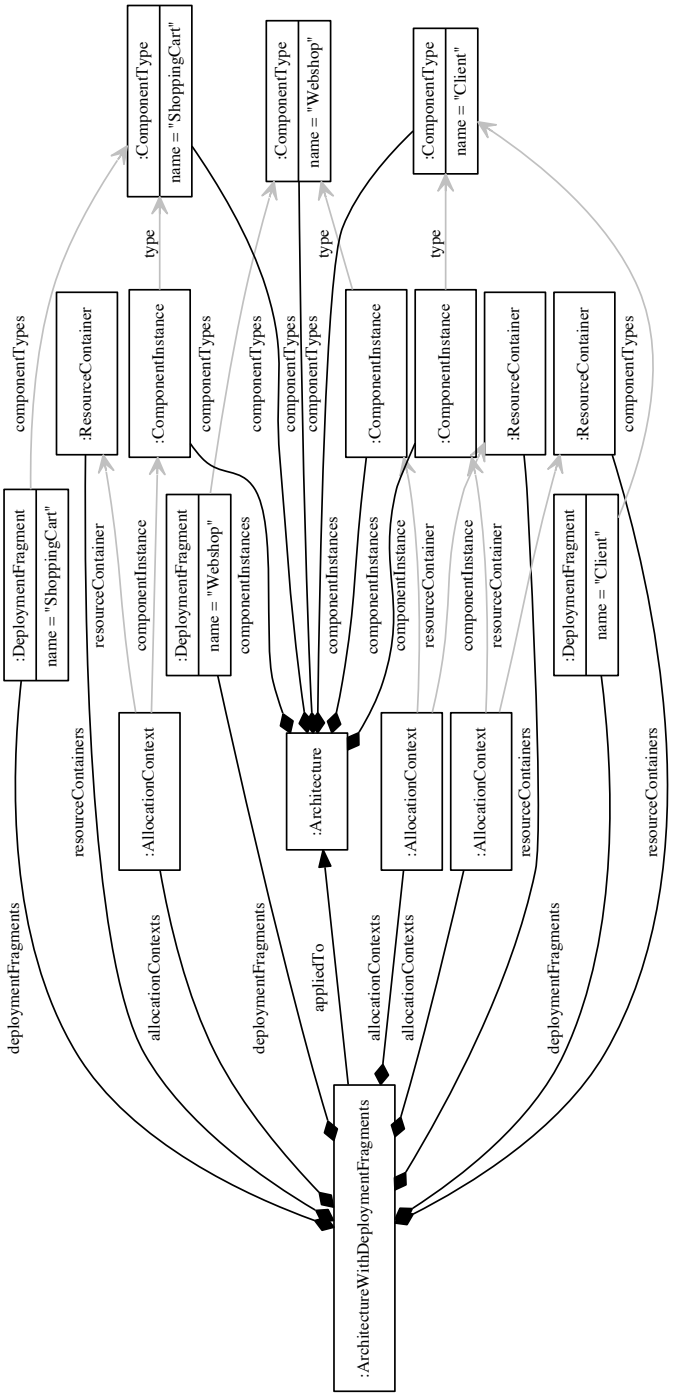
The stereotypes are applied as follows:

$$\text{ArchitectureWithDeploymentFragments} \xrightarrow{appliedTo} o_{Architecture}$$

The following values and targets are assigned to attributes and references:

$$\begin{aligned} &o_{Architecture}.\text{deploymentFragments} \xrightarrow{references} o^{ShoppingCart}_{DeploymentFragment}, \\ &o_{Architecture}.\text{deploymentFragments} \xrightarrow{references} o^{Webshop}_{DeploymentFragment}, \\ &o_{Architecture}.\text{deploymentFragments} \xrightarrow{references} o^{Client}_{DeploymentFragment}, \\ &o^{ShoppingCart}_{DeploymentFragment}.\text{componentTypes} \xrightarrow{references} \text{ShoppingCart}, \\ &o^{Webshop}_{DeploymentFragment}.\text{componentTypes} \xrightarrow{references} \text{Webshop}, \\ &o^{Client}_{DeploymentFragment}.\text{componentTypes} \xrightarrow{references} \text{Client}, \\ &o_{Architecture}.\text{resourceContainers} \xrightarrow{references} o^{Server1}_{ResourceContainer}, \\ &o_{Architecture}.\text{resourceContainers} \xrightarrow{references} o^{Server2}_{ResourceContainer}, \\ &o_{Architecture}.\text{resourceContainers} \xrightarrow{references} o^{Server3}_{ResourceContainer}, \\ &o_{Architecture}.\text{allocationContexts} \xrightarrow{references} o^{1}_{AllocationContext}, \end{aligned}$$

$$o_{Architecture}.\text{allocationContexts} \xrightarrow{references} o^{2}_{AllocationContext},$$
$$o_{Architecture}.\text{allocationContexts} \xrightarrow{references} o^{3}_{AllocationContext},$$
$$o^{1}_{AllocationContext}.\text{resourceContainer} \xrightarrow{references} o^{Server1}_{ResourceContainer},$$
$$o^{1}_{AllocationContext}.\text{componentInstance} \xrightarrow{references} o^{ShoppingCart}_{ComponentInstance},$$
$$o^{2}_{AllocationContext}.\text{resourceContainer} \xrightarrow{references} o^{Server2}_{ResourceContainer},$$
$$o^{2}_{AllocationContext}.\text{componentInstance} \xrightarrow{references} o^{Webshop}_{ComponentInstance},$$
$$o^{3}_{AllocationContext}.\text{resourceContainer} \xrightarrow{references} o^{Server3}_{ResourceContainer},$$
$$o^{3}_{AllocationContext}.\text{componentInstance} \xrightarrow{references} o^{Client}_{ComponentInstance}$$


:ComponentType
name = "ShoppingCart"
:ComponentType
name = "Webshop"
:ComponentType
name = "Client"
type
type
type
componentTypes
componentTypes
componentTypes

C.23 Namespace

Figure C.26 shows an example model of the profile *Namespace*. Example 34 gives the formal definition of this model.

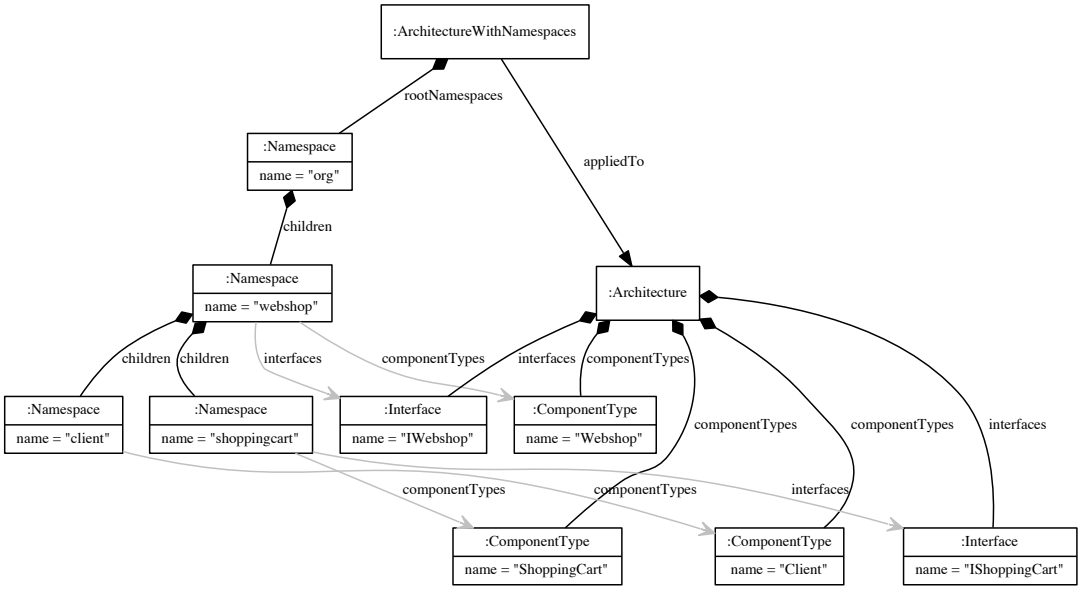


Figure C.26: Example application of the profile *Namespace*

Example 34: Example application of the profile *Namespace*

The exemplary profile application $M^{Example}_{Namespace}$, that instantiates the profile *Namespace*, is formalized as follows. Figure C.26 accompanies the definition as an overview.

$$P = P_{Namespace}$$

$$R = \{M^{Example}_{Kernel}\}$$

$$O := \{o_{org}, o_{webshop}, o_{shoppingcart}, o_{client}\}$$

The elements are named as follows:

$$name(o_{org}) = \text{org},$$
$$name(o_{webshop}) = \text{webshop},$$
$$name(o_{shoppingcart}) = \text{shoppingcart},$$
$$name(o_{client}) = \text{client}$$

The stereotypes are applied as follows:

$$\text{ArchitectureWithNamespaces} \xrightarrow{appliedTo} o_{Architecture}$$

The following values and targets are assigned to attributes and references:

$$o_{Architecture}.\text{rootNamespaces} \xrightarrow{references} \text{org},$$
$$\text{org.children} \xrightarrow{references} \text{webshop},$$
$$\text{webshop.interfaces} \xrightarrow{references} \text{IWebshop},$$
$$\text{webshop.componentTypes} \xrightarrow{references} \text{Webshop},$$
$$\text{webshop.children} \xrightarrow{references} \text{shoppingcart},$$
$$\text{shoppingcart.interfaces} \xrightarrow{references} \text{IShoppingCart},$$
$$\text{shoppingcart.componentTypes} \xrightarrow{references} \text{ShoppingCart},$$
$$\text{webshop.children} \xrightarrow{references} \text{client},$$
$$\text{client.componentTypes} \xrightarrow{references} \text{Client}$$

C.24 Secure Information Flow

Figure C.27 shows an example model of the profile *Secure Information Flow*. Example 35 gives the formal definition of this model.

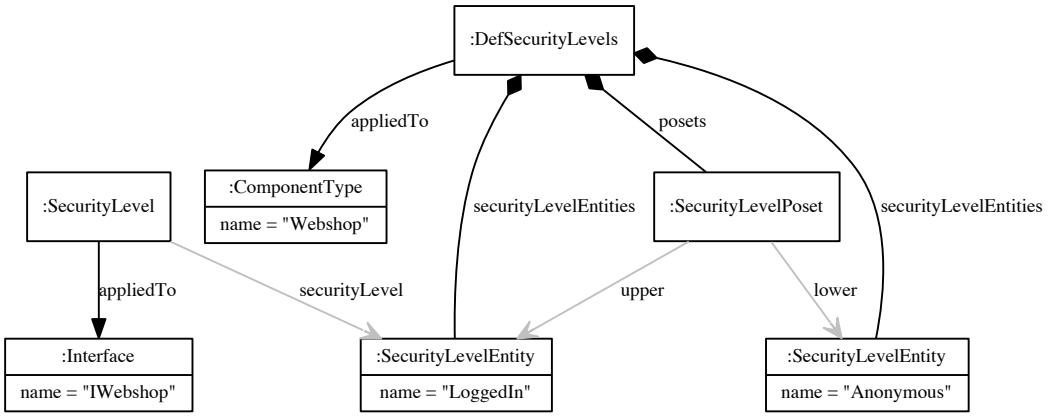


Figure C.27: Example application of the profile *Secure Information Flow*

Example 35: Example application of the profile *Secure Information Flow*

The exemplary profile application $M^{Example}_{SecureInformationFlow}$, that instantiates the profile *Secure Information Flow*, is formalized as follows. Figure C.27 accompanies the definition as an overview.

$$P = P_{SecureInformationFlow}$$

$$R = \{M^{Example}_{Kernel}\}$$

$$O := \{o_{SecurityLevelPoset}, o_{LoggedIn}, o_{Anonymous}\}$$

The elements are named as follows:

$$name(o_{LoggedIn}) = \text{LoggedIn},$$
$$name(o_{Anonymous}) = \text{Anonymous}$$

The stereotypes are applied as follows:

$$\text{DefSecurityLevels} \xrightarrow{appliedTo} \text{Webshop},$$
$$\text{SecurityLevel} \xrightarrow{appliedTo} \text{IWebshop}$$

The following values and targets are assigned to attributes and references:

$$o_{SecurityLevelPoset}.\text{upper} \xrightarrow{references} \text{LoggedIn},$$
$$o_{SecurityLevelPoset}.\text{lower} \xrightarrow{references} \text{Anonymous},$$
$$\text{Webshop.posets} \xrightarrow{references} \text{SecurityLevelPoset},$$
$$\text{Webshop.securityLevelEntities} \xrightarrow{references} \text{LoggedIn},$$
$$\text{Webshop.securityLevelEntities} \xrightarrow{references} \text{Anonymous},$$
$$\text{IWebshop.securityLevel} \xrightarrow{references} \text{LoggedIn}$$

C.25 Time Resource Demand

Figure C.28 shows an example model of the profile *Time Resource Demand*. Example 36 gives the formal definition of this model.

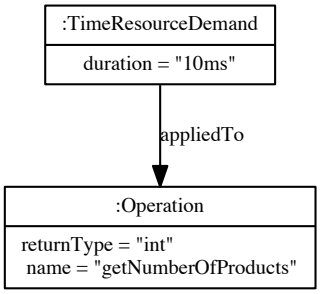


Figure C.28: Example application of the profile *Time Resource Demand*

Example 36: Example application of the profile *Time Resource Demand*

The exemplary profile application $M^{Example}_{TimeResourceDemand}$, that instantiates the profile *Time Resource Demand*, is formalized as follows. Figure C.28 accompanies the definition as an overview.

$$P = P_{TimeResourceDemand}$$

$$R = \{M^{Example}_{OperationInterfaces}\}$$

The stereotypes are applied as follows:

$$\text{TimeResourceDemand} \xrightarrow{appliedTo} \text{getNumberOfProducts}$$

The following values and targets are assigned to attributes and references:

$$\text{getNumberOfProducts.duration} \xrightarrow{hasValue} \text{10ms}$$

List of Listings

List of Figures

List of Integration Mechanisms

List of Definitions

List of Constraints

List of Examples